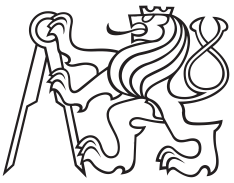
Diplomová práce



České
vysoké
učení technické
v Praze

F3

Fakulta elektrotechnická
Katedra řídicí techniky

Zpracování dat naměřených elektromagnetickým spektrálním analyzátorem IZ225

Vojtěch Kůrka
Otevřená informatika - Počítačové inženýrství

Květen 2016
Vedoucí práce: Dr. Ing. Alexandru Moucha

/ Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze dne 21. 5. 2016

.....

Abstrakt / Abstract

Tato práce vznikla na základě požadavku rozšířit program vytvořený při mé bakalářské práci, jenž zobrazoval data z digitálního příjímače IZ225 vyvíjeného ve firmě Intriple a.s.. Program je napsán v C++ a vyvíjen v Embarcadero Studio XE7. Nová funkcionalita má za úkol zvýšit komfort při práci s daty a umožnit jejich hlubší a částečně automatizovanou analýzu.

This final project is based on demand to expand the program created during my bachelor thesis, which controlled and displayed data from the digital reciever IZ225 developed by the company Intriple a.s.. Program is written in C++ using Embarcadero Studio XE7. Result of the new functionality should introduce easier way to work with the received data and allowing its deeper and partially automated analysis.

Obsah

Tabulky / Obrázky

Kapitola 1
Úvod

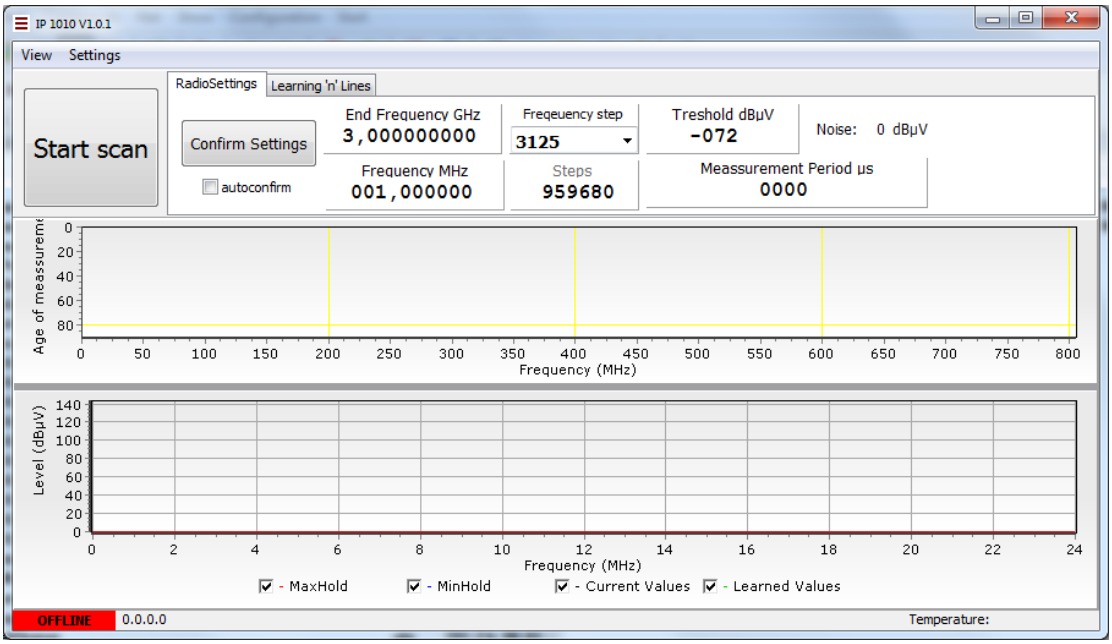
V prvním ročníku bakalářského studia jsem nastoupil do firmy Intriple a.s., jako programátor, abych získal praxi, kterou zaměstnavatelé vyžadují a vylepšil svůj osobní rozpočet. Při volbě tématu bakalářské práce pak přišel nápad spojit příjemné s užitečným a tak jsem zaměstnavatele oslovil, zda by se nenašel nějaký vhodný projekt, který by byl k tomuto účelu využitelný. A tak vznikla moje bakalářská práce.

V té jsem měl za úkol navrhnout program pro vizualizaci dat ze scanneru `IZ225`. Chopil jsem se toho úkolu a pod rukama mi tak začal vznikat můj první větší projekt. Výsledky se dostavovaly poměrně rychle a já měl radost ze zprovoznění každé dílčí části - příjmu dat, ovládání přijímače, vykreslení dat,...

Výsledkem byl funkční program, který se ve firmě používal a stále používá nejen při testování funkčnosti přijímače. Občas se vyskytl požadavek na přidělání nějaké drobné funkcionality, která by usnadnila práci. Následně vznikl požadavek na větší rozšíření a já se chytil příležitosti a proměnil rozšíření v zadání diplomové práce.

Původní ‚dobastlování' nové funkcionality nebyl problém, jelikož šlo o drobnosti. Při konfrontaci s novým zadáním se však začaly projevovat návrhové chyby a při větších zásazích i chyby, které byly implementační, ale skryté (mezi ně patřilo například neuvolňování paměti).

Při rozšiřování tak musely být adresovány nejdříve tyto vady a vyřešeny s ohledem na rozšiřování, které je známo nyní, ale i na to budoucí, jehož specifikace teprve přijde.



Obrázek 1.1. Podoba programu z bakalářské práce.

Kapitola 2
Vylepšení a optimalizace stavu programu

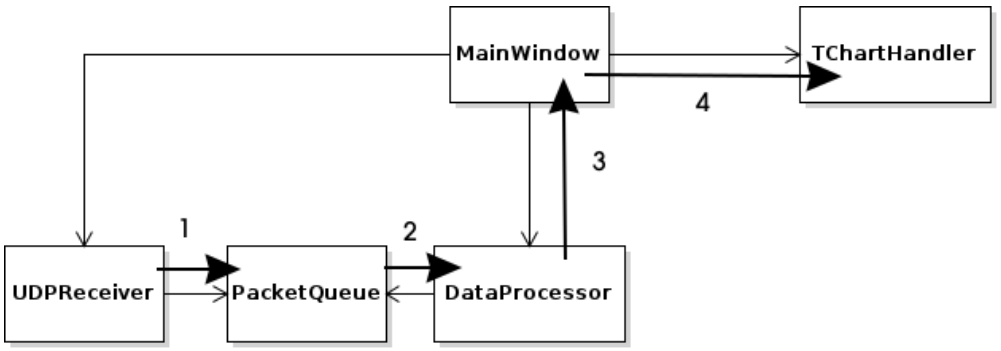
Jak jsem již uvedl v úvodu, v rámci bakalářské práce jsem dosáhl toho, že je možné s digitálním přijímačem pracovat. Je možné přístroj nastavit, spustit měření, přijmout data a zobrazit je v uživatelsky přívětivé podobě.

Kvůli své nezkušenosti jsem však při návrhu toho, jak data procházejí programem udělal několik chyb. První byla nedostatečná granularita tříd a jejich oddělení. To se projevilo v momentě, kdy jsem se snažil program rozšířit. Nejdříve se tedy pustím do napravení svých starých hříchů.

2.1 Zobrazení

Ukázkovým příkladem chybně navržené třídy byl DataProcessor. Tato třída se stará o vyzvedávání paketů z příchozí fronty a jejich skládání v jedno konkrétní měření. Kromě toho, ještě toto měření připravuje obrazová data pro vodopád a aktualizuje hodnoty ve spektru. To je kombinace, která spolu sice souvisí, ale výrazně ubírá na čitelnosti a srozumitelnosti kódu, jelikož jde o činností příliš mnoho.

Podívám se tedy nejdříve na to, jak vypadá situace a jaký je tok dat. Ten je znázorněn na obrázku 2.1 šipkami. Data jsou přijímána v `UDPReceiver`, přes `PacketQueue` se dostanou do `DataProcessor`, kde se skutečně celá zpracují. V MainWindow běží smyčka, která se jednou za 100 ms dotáže na nově spočítaná data zobrazovačů. Spektrum se vykreslí napřímo a data pro vodopád ještě putují skrz TChartHandler.



Obrázek 2.1. Původní návrh přijímání a zpracování dat.

Do takovéhoto molochu, jakým `DataProcessor` bezpochyby je, je velmi těžké něco přidávat a je tedy nutné tuto funkcionalitu rozdělit do více tříd, provádějící v ideálním případě jednu věc a pořádně. Nový návrh by měl dovolovat co největší separaci ovládání přijímače, příjmu packetů a zobrazení dat.

Prvním krokem bude ořezání schopností `DataProcesoru` na jednoduché vyzvednutí packetu z fronty a sestavení měření. Toto sestavené měření by se následně mělo předat dále a nějakým způsobem zobrazit.

Nyní vyvstala otázka, jak zobrazit zpracovaná data. Mám vodopád, spektrum, má přibýt ještě zobrazení v čase, ukládání a detektor. Těchto několik zobrazovačů (ke kterým jistě v budoucnu ještě něco přibyde) by bylo velmi nešťastné referencovat přímo z `DataProcessor`, došlo by totiž k opětovnému nežádoucímu zesložitění třídy. Ideálním řešením se tak jeví vytvořit další třídu, která bude mezi zpracováním dat a zobrazovači. Funkčně tak bude tato třída zajišťovat pouze to, že vezme data od `DataProcessoru` a následně je předá všem aktivním zobrazovačům.

Jelikož zobrazovaču může být více a dokonce se může objevovat několik vícero různě nakonfigurovaných instancí toho samého, tak vytvářet speciální funkce pro přístup ke každému z nich spolu s nějakou formou uchovávání jejich instancí není vhodné. Je nutné adresovat tento problém a nad správou zobrazovačů se zamyslet. Správa by měla být co nejjednodušší, aby tento nedocházelo ke zbytečnému narůstání kódu a bylo umožněno snadno přidat další typ zobrazovače.

Minimalizovat režii práce se zobrazovači mi dovolí použití abstraktní třídy. Pokud každý zobrazovač bude potomkem této třídy, dojde ke sjednocení jejich rozhraní a práce s nimi se tak omezí na několik málo funkcí. K uchovávání instancí bude moci být použit například `std::vector`. A předání měření se zredukuje na projití jednoho for-cyklu.

Protože je vlastní měření v podstatě jen pole bajtů, bude k jejich správnému zobrazení potřeba doplnit další informace. K tomu poslouží struktura v tabulce 2.1. Uložím do ní všechny podstatné informace o vlastnostech měření a nebude tak problém s daty dále pracovat.

Typ	Název pole	Popis
uint64_t	StartFrequency	První naměřená frekvence
uint64_t	StopFrequency	Poslední naměřená frekvence
uint32_t	StepCount	Počet naměřených hodnot
uint32_t	StepSize	Vzdálenost naměřených bodů
int32_t	NoiseLevel	Vypočítaná úroveň šumu
uint32_t	Period	Nastavený časový rozestup mezi měřeními
double	ArrivalTime	Čas přijetí prvního packetu měření

Tabulka 2.1. Parametry měření

2.1.1 TDisplay

Nejdříve je potřeba vytvořit mustr, podle kterého se budou zobrazovače tvořit. K tomu poslouží třída `TDisplay`. Protože jde o grafický prvek, bude vhodné, aby třída byla potomkem `TForm`. Tím získám možnost, kromě nadefinování vlastního grafického zobrazení, ještě použití WYSIWYG editoru dostupného ve studiu.

Projekt využívá grafické komponenty VCL. Ty jsou implementovány v jazyce Object Pascal a jsou dostupné přímo ve studiu. Problémem využití těchto komponent je, že není možné upravovat jejich parametry z jiného než z hlavního vlákna. Nejsou thread-safe. Zavoláním z jiného vlákna se tak může vykonat požadovaná akce, nebo shodit celý program.

Z hlediska výkonu a uživatelského komfortu, je pak příhodné, že `DataProcessor` pracuje ve vlastním vlákně. Přináší to s sebou však kopmplikaci, právě kvůli použití VCL komponent a nutnosti pracovat s nimi pouze z hlavního vlákna. Z toho důvodu bude muset být odděleno zpracování dat a jejich vykreslení. `TDisplay`, tedy musí obsahovat funkci pro zpracování a další pro vykreslení dat, které musejí být nezávislé na sobě.

Dále je zde učení, to musí být možné zobrazit a jelikož minimálně pro spektrum je to nezbytnost, musí zde být metoda pro předání naučené hodnoty zobrazovačům.

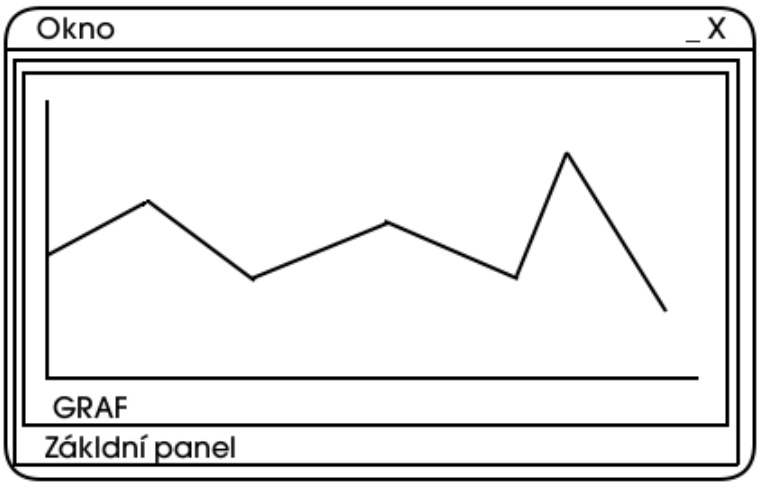
Kvůli spektru je nutné zahrnout metody pro nastavení barevného rozsahu. Toto nastavení však bude vhodné aplikovat na všechny zobrazovače. Na spektru se tím upraví rozsah úrovní a stejně tak i na časovém průběhu.

`TForm` je ze své podstaty okno. Já však chci mít možnost vodopád a spektrum zobrazit v hlavním oknu, ale zároveň je potřeba některé zobrazovače zobrazit v okně samostatném. Zde půjde využít toho principu, že okno `TForm` po vytvoření, není nutné zobrazit metodou `Show()`, ale i přes to budou prvky v něm obsažené inicializovány a budou normálně fungovat. A toho principu, že každému prvku lze nastavit rodiče upravením hodnoty `Parent`.

Parametrem této property je jiná VCL komponenta. Efekt tohoto přiřazení je pak ten, že komponenta, které byl takto změněn rodič se začne vykreslovat na takto nadefinovaném rodiči. Pokud tedy tlačítku `TButton` přiřadím jako rodiče nějakou instanci `TPanelu`, začne se tlačítko na příslušném panelu vykreslovat. Stejně tak se stane s panelem, kterému jako rodiče nastavím jiný panel. Panel se změněným rodičem se začne vykreslovat na novém rodičovském panelu. A protože tento princip platí obecně. Jakýkoliv grafický prvek, který má nastavený jako rodiče tento panel se změněným rodičem, se bude vykreslovat na stejném panelu.

Pokud každý zobrazovač bude navrhován tak, že se na okně vytvoří jeden panel sloužící jako podklad pro všechny ostatní prvky, bude se moci tomuto panelu přiřadit jakýkoliv panel a umístit tak jeho obsah kamkoliv bude potřeba. Proto rozhraní musí bude obsahovat metodu pro předání panelu, který bude použit jako rodič.

Návrh spektru v tomto režimu pak může vypadat jako na obrázku 2.2.



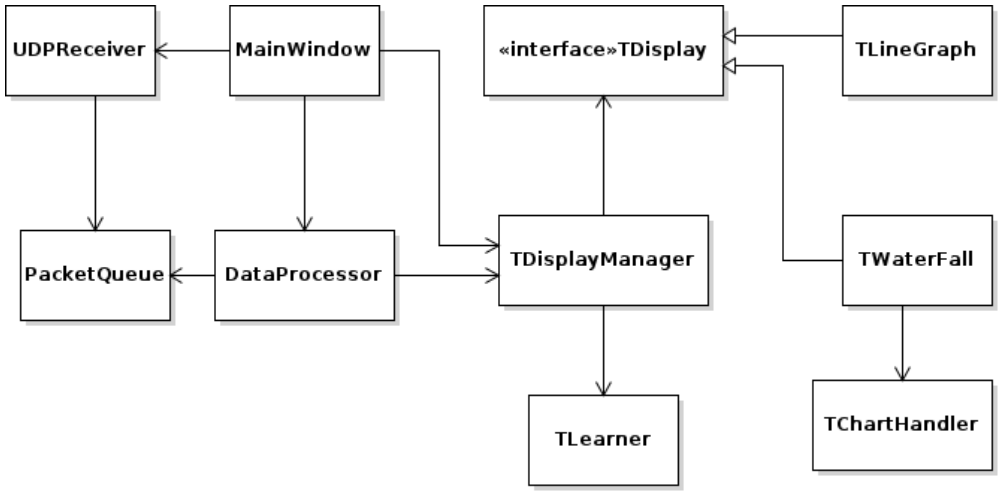
Obrázek 2.2. Princip návrhu okna přijímače

U zobrazovače je také potřeba mít možnost zbavit se zobrazovaného obsahu. K tomu poslouží funkce clear. Celé rozhraní pak vypadá takto:

```
class TDisplay : public TForm{
protected:
    ...
public:
    __fastcall TDisplay(TComponent* Owner);
    __fastcall virtual ~TDisplay(){};
    virtual void Delete(){};
    virtual void AddSweep(SweepConfiguration SweepConf,
            unsigned char * data)=0;
    virtual void SweepRepaint()=0;
    virtual void Clear(){};
    virtual void RefreshLearn(unsigned char * LearnLine){};
    __property TPanel * BasePanel = {read=FBasePanel,
                                     write=SetBasePanel};
    __property int MinLevel = {read=FMinLevel, write=SetMinLevel};
    __property int MaxLevel = {read=FMaxLevel, write=SetMaxLevel};
};
```

2.1.2 TDisplayManager

Když je vzor pro zobrazovače hotový, je na čase vyřešit jak s nimi bude pracováno. K tomu bude sloužit `TDisplayManager`. Jeho úkolem bude pracovat jako proxy pro distribuci přijatých měření, vytváření nových oken a celkově práci se zobrazovači tak, aby od nich byla zbylá část programu oddělena a struktura programu pro zpracování dat by měla vypadat jako na obrázku 2.3.



Obrázek 2.3. Přepracované přijímání a zpracování dat.

Přijatá data jsou z `DataProcessor`, kde je z nich poskládáno měření, předána `TDisplayManageru` a ten je distribuuje zobrazovačům. V nich dojde ke zpracování dat a následně jejich vykreslení. Třída `TLearner`, pak bude sloužit, jak již název napovídá, k obstarání funkčnosti učení.

Překreslení probíhá na základě zavolání funkce `RepaintDisplays()`. Ta je periodicky volána z hlavního okna pomocí časovače.

`TDisplayManager` se principielně chová jako zobrazovač `TDisplay` - jsou mu předána data, která následně zobrazí prostředníctvím dalších zobrazovačů. Z toho důvodu je vhodné vycházet při návrhu rozhraní, které bude poskytovat, z návrhu `TDisplay`. Z něho přeberu `AddSweep()`, `SweepRepaint()`, `Clear()` a funkce pro nastavení Min a Max Level. Tyto budou pro manažer znamenat, že se zavolají na všech aktuálně používaných zobrazovačích.

Aby bylo možné vytvořit zobrazovač, je pro každý potřeba vytvořit vlastní funkci na vytvoření. Protože je možné jej vytvořit v samostatném okně, nebo na panelu, musí tato metoda akceptovat ukazatel na panel, na kterém se má vytvořit. Vytváření spektra tak bude vypadat takto:

```
TLineGraph * TDisplayManager::CreateLineGraph(TPanel * Panel){
    TLineGraph * LineGraph;
    if(Panel == NULL){
        LineGraph = new TLineGraph(NULL);
        LineGraph->Show();
    }else{
        LineGraph = new TLineGraph(NULL);
        LineGraph->BasePanel = Panel;
    }
    if(LineGraph != NULL) FDisplays.push_back(LineGraph);
    return  LineGraph;
}
```

Pokud nedostanu validní pointer na `TPanel` (tedy jiný než `NULL`), vytvořím `TLineGraph` a přiřadím mu daný pointer. Tím se veškerý obsah, který jsem v editoru umístil na základní panel zobrazí na panelu, na který je ukazováno. Pokud funkce dostane `NULL`, je okno zobrazeno metodou `TForm - Show()`. Následuje zařazení do vectoru mezi ostatní displeje a vrácení vytvořené instance pro případ, že je potřeba nějakých další úprav.

Problém nastává, když je zobrazovač v v samostatném okně a to je zavřeno. `TForm` disponuje událostí `OnClose()`, která je vyvolaná při zavření okna a v té by mohlo dojít k odstranění instance. A zde je právě ten problém, instance okna zmizí, avšak zobrazovač je stále referencován z manažeru. Je mu tedy potřeba před samotným odstraněním instance zobrazovače nějak sdělit, že si nějakou instanci nemá dále obhospodařovat.

Protože zobrazovač přímo nemá jak zajistit předání této informace, využiji fronty `EventQueue`. Typ události je `DELETEDISPLAY` a použitý typ bude pouze pointer. Do toho bude uložen ukazatal na instanci, která si přeje být smazána. Na základě této informace dojde v hlavním okně k zavolání funkce `DeleteDisplay(TDisplay * DisplayToDelete)`, ve které se provede bezpečné odebrání z vektoru a odstranění instance.

Jakmile jsou některé zobrazovače připraveny, může manažer vykonávat svou činnost. Po příjmu dat z `DataProcessoru` dochází k předzpracování dat. O tom však napíši v podkapitole 2.1.3. Předzpracování dat.

2.1.3 Předzpracování dat

Zpracování dat třídou `TLearner`. Ta má 4 režimy ve kterých může pracovat a to v závislosti na tom, zda je, či není aktivováno učení a jeho aplikace na naměřené hodnoty.

Učení je prováděno formou maxholdu, kdy se pro každou frekvenci zaznamenává největší hodnota, které na ní bylo dosaženo. Je-li aktivní aplikování učení dochází k nastavení hodnot nižších než té, která je na dané frekvenci naučená. Efektivně jsou tak odfiltrovány všechny nezajímavé výsledky.

V případě, že je aktivní jak učení, tak jeho aplikace, dochází nejdříve k aplikaci naučených hodnot a teprve po té k učení. Pokud by to bylo obráceně, nebylo by nic vidět, protože naučená hodnota by nikdy nebyla menší, než hodnota měření a všechny by tak byly nastaveny do úrovně šumu a například vodopád by byl celý modrý.

Kromě učení dochází ještě k předzpracování pomocí `TSweepAggregator`. Proč zde je rozvedeno v kapitole o nahrávání. Jeho funkce je však prostá a jeho rozhraní se skládá v podstatě jen ze dvou funkcí:

```
class TSweepAggregator{
    public:
        TSweepAggregator();
        ~TSweepAggregator();
        unsigned char * AddLine(SweepConfiguration Conf, unsigned char *
        line);
        __property double Aggregation = {write=SetAggregation,
        read=FAggregation};
};
```

Cílem je zpomalit tok dat při předávání displejům. Jak moc je tok zpomalen, je ovládáno pomocí property double. Ta určuje, jak často je měření puštěno dál. Nastavuje se hodnotou v sekundách. Pro hodnotu 0,5 tak bude předávat měření dvakrát za sekundu.

Zpomalování je realizováno přes `AddLine()` přijímající měření podobně jako zobrazovače. Výstupem této funkce je na rozdíl od `void` zobrazovačů `unsigned char *`. Uvnitř této třídy je totiž buffer, který velikostně odpovídá měření. Ten je plněn principem maxhold, tak aby uniklo co nejméně informací. Na zavolání odpovídá navrácením hodnoty `NULL`. Jakmile je však překročena zadaná doba, dojde k předání bufferu a ten je následně zpracován zobrazovači. Buffer se resetuje a držení maxholdu začíná nanovo.

Tento agregátor zde musí být ve dvou instancích, první zpracovává data zobrazovače a druhý speciálně pro nahrávání. To je z důvodu delší absence při nahrávání měření. Je tak možné nastavit nahrávací agregaci na půl sekundy a zobrazovací na například 10 a získat tak přehled o tom, co se nahrálo.

2.2 TWaterfall

Kód vodopádu, který byl dosud součástí `DataProcesor` a hlavního okna bylo potřeba přesunout. Umístil jsem jej tedy do samostatné třídy `TWaterfall`, která je potomkem `TDisplay`. V něm je navržené grafické rozhraní a přebírání dat. Samotné zpracování pak probíhá v `TWaterfallController`. V `DataProcessor` docházelo ke ztrátové kompresi dat, kvůli rychlosti překreslování. Ta vypadala tak, že se zjistilo, jak široký je zobrazovaný řádek v pixelech a na tuto velikost se napřijímaná data přepočítala. Ztratila se tím tedy jemnost zobrazení. Řádek mohl mít až téměř milion bodů, na displej se však vešlo jen cca tisíc.

Nyní se spolu se strukturou zobrazovačům předává měření nezměněno a je pouze na nich, jak se s takto velkým počtem dat vypořádají. U vodopádu nejdéle trvá zobrazení, které je dosahováno přiohnutím účelu grafu `TChart` pomocí `TChartHandleru`. Přiohnutí spočívá ve vytvoření dummy série, aby se zobrazily osy grafu a na základě těchto os dochází k promítání vodopádu na pozadí grafu. Toto promítnutí na pozadí není dělané pro časté aktualizace, protože se počítá s tím, že se nahraje jeden rozumně velký obrázek, do kterého se už nebude sahat. Při rozměrech obrázku vodopádu blížícím se cca 40 000x1000 pixelů tak překreslení trvalo velmi dlouho, nad touto hranicí se pak už nevykreslil vůbec.

Protože jsem potřeboval, aby se žádná data neztrácela a bylo možné si prohlédnout každý jednotlivý bod měření (tedy i každý bod z obrázku o rozměrech 900 000x1000), bylo nutné přistoupit k předzpracování.

To vypadá tak, že mám kruhový buffer pro jednotlivá napřijímaná měření, který s každým zavoláním `AddSweep()` aktualizuji tak, aby poslední přijatý řádek byl vždy na začátku. Tento buffer s každým překreslením předávám do `TChartHandleru` ve kterém dochází k přepočítání na rozměry shodné s těmi, které bude mít výsledný obrázek na pozadí.

Toto přepočítávání pak bylo nutné optimalizovat. První optimalizací bylo právě využití kruhového bufferu pro ukládání dat namísto ukládání v souvislém poli. Při přidání řádku se tak přesune pouze pole pointerů namísto obrovského pole desítem megabajtů dat.

Přechod na vyšší verzi IDE kromě jiných vylepšení přinesl i možnost použít `TThreadPool`. Jde o klasický threadpool, který poskytuje možnost rychlé paralelizace úkonů bez nutnosti se starat o vytváření, udržování a ukončování vláken. Princip použití je jednoduchý. Na začátku si vytvořím instanci `TThreadPool` a nastavím, kolik vláken má být použito. Z principu fungování threadpool, není potřeba víc než jeden nastaven na celočíselný násobek počtu vláken, než je počet jader procesoru. Já používám dvojnásobek.

Z nepřeberného množství způsobů[1] jakým lze `TThreadPoolu` předat úlohu na zpracování jsem zvolil ten pravděpobně nejjednodušší:

```
static TLoopResult __fastcall For(System::TObject* Sender, int
  ALowInclusive, int AHighInclusive, TIteratorEvent AIteratorEvent);
```

Jeho myšlenkou je paralelizace for-cyklu tak, že při každé iteraci dojde k zavolání funkce `AIteratorEvent`. Iteruje se po jedné a to od hodnoty `ALowInclusive` až po `AHighInclusive`. `TObject *Sender` je standardní konstrukce pro zpracování událostí při práci s VCL, odkazuje na objekt, který událost vyvolal. Zde žádný není a tak jej nastavím do `NULL`.

For-cyklus, který paralelizuji je koncipován tak, že v každém cyklu přepočítávám jeden řádek vodopádu, který se chystám zobrazit.

Poslední optimalizací, která však na výkon měla největší vliv, bylo přidání bufferu do `TChartHandleru`. V něm jsou uchovávány již přepočítané řádky podobně, jako originály v `TWaterfallControlleru`. Protože k datům nepřidávám žádnou informaci o posunu od dalšího vykreslování, spoléhám se při rozhodování o tom, které řádky přepočítat na ukazatele. V `TChartHandleru` si při překreslení poznamenám, které pole představovalo první řádek. Protože je používám opakovaně, stačí si přepočítat řádky mezi aktuálně prvním a tím, který byl první předtím.

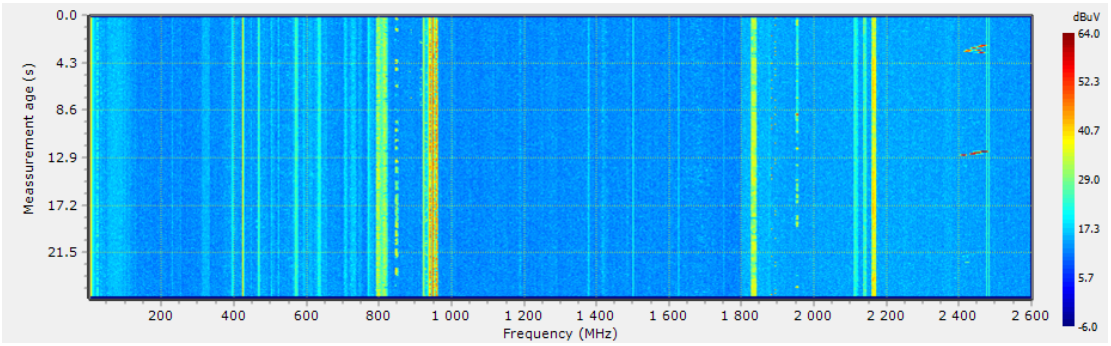
Při velkých měřeních, které trvají relativně dlouho tak musím přepočítat jen pár nově přidaných řádek, namísto celého vodopádu majícího řádků cca tisíc.

Proces překreslení vodpádu se tak zkrátil z cca 50 ms na 3 ms.

Pro zvýšení uživatelské přívětivosti je rovněž přepracována levá osa. Dříve se na ni zobrazovalo kolikátý řádek měření je zobrazován. Nyní jde o relativní stáří vztažené k nejvyššímu řádku. K docílení tomuto mi dopomohla přítomnost času přijetí měření ve struktuře `SweepConfiguration`. Podobně jako jednotlivé řádky i tyto časy uchovávám v kruhovém bufferu. Při překreslení, pak vezmu rozdíl přijetí každého čtyřicátého řádku od prvního a doplním na levou osu.

`TChartHandler` starající se o překreslení se kromě výkonové optimalizace dočkal ještě jednoho vylepšení. Tím je systém vlastních komponent. Ty jsou spravovány podobně jako zobrazovače. Mají předka, poskytující jednotné rozhraní na jehož základu jsou tvořeny. Rozhraní je tvořeno převážně zpracováním uživatelských událostí - kliknutí myší, stisk klávesnice... Rozdíl oproti zobrazovačům je ten, že tyto komponenty mají obvykle specifickou funkci, která je svázána s grafem, nad kterým `TChartHandler` pracuje a tím pádem je často nutné používat unikátní veřejné funkce pro danou komponentu.

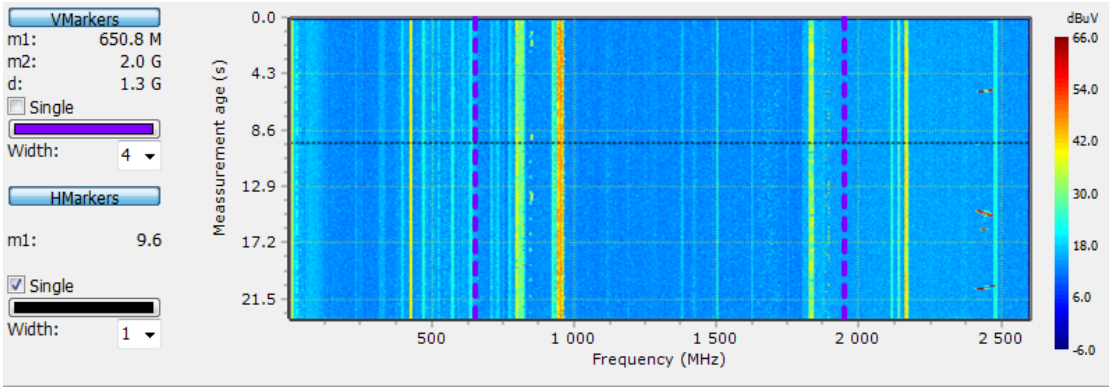
První je paleta viz obrázek 2.4. Tato pomáhá uživateli udělat si lepší obrázek o tom, jakých konkrétních hodnot dosahují úrovně naměřených signálů. Barevný graf palety je kreslen pomocí `TBitmap`. Tato VCL komponenta je rovněž použita pro pozadí grafu. Díky tomu, je možné k jednotlivým pixelům přistupovat stejně.



Obrázek 2.4. Paleta pro snazší rozpoznávání úrovní signálu.

Po nastavení rozmezí které je zobrazeno dojde v `TChartHandleru` k vygenerování palety, která je následně používána pro kreslení vodopádu. Tato paleta funguje tak, že mapuje naměřené úrovně uložené v osmi bitech na 24bitovou RGB barvu a velmi tak usnadňuje generování obrázku. Kromě použití palety pro samotná obrázek vodopádu je však paleta předána i této komponentě spolu s rozsahem, nad kterým se rozpíná. Do bitmapy komponenty pak lineárně promítnu tento rozsah, čímž získám barevný graf a dopočítáním pěti mezihodnot jeho popisky.

Další komponentou jsou markery (Obrázek 2.5). Těch je možné využít pro zjištění délky trvání a šířky signálu. Markery jsou zobrazeny po stisknutí tlačítka označeného Markers. Výchozí poloha je vždy ve 25% a 75% zobrazovaného intervalu. Vlevo je panel, na kterém jsou pro svislé i vodorovné zobrazeny pozice a jejich rozdíl. Ovládání je jednoduché. Stačí kliknout a táhnout. Na panelu je kromě hodnot zobtazena i barva a šířka, kterou lze modifikovat. Takto provedené změny jsou ukládány do ini souboru k ostatním nastavením a změny jsou zachována i pro příští spuštění programu.



Obrázek 2.5. Markery pro měření ve vodopádu.

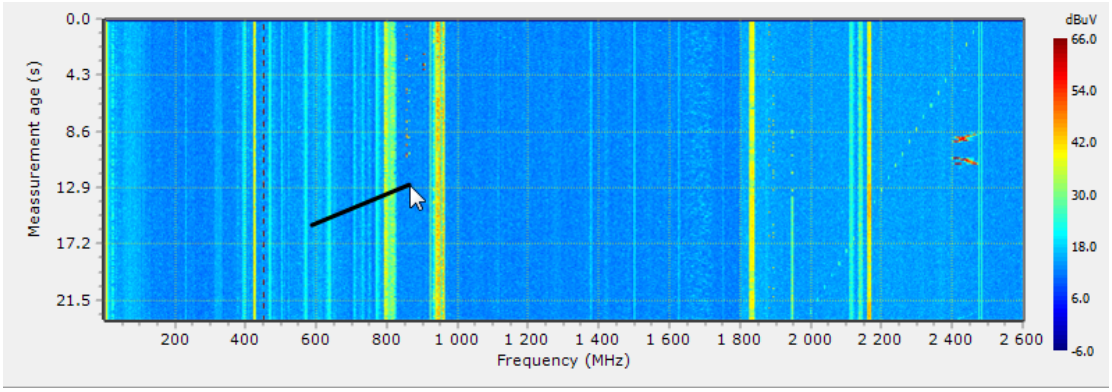
K ovládání slouží myš. Po zobrazení markerů stačí kliknout marker a táhnout, aby ho uživatel přesunul. Pro větší přívětivost, pak jakmile na marker najede kurzorem, změní se na obousměrnou šipku znázorňující směr, kterým lze marker posunout. Pozice a rozdíl, v případě zobrazení obou, jsou zobrazeny na panelu vlevo.

Markery jsou realizovány pomocí série, běžně sloužící ke kreslení grafů. Po kliknutím levého talčítka na marker dojde s každým vyvoláním funkce `OnMouseMove()` k jeho překreslení na pozici kurzoru a zobrazení pozice. Panel vlevo, na kterém jsou informace zobrazovány je vytvořen při návrhu a předáván podobně jako je to při umisťování zobrazovače. Protože jsou tyto události vyvolávány událostmi grafického rozhraní, není potřeba kontrolovat přístup k úpravám grafického rozhraní.

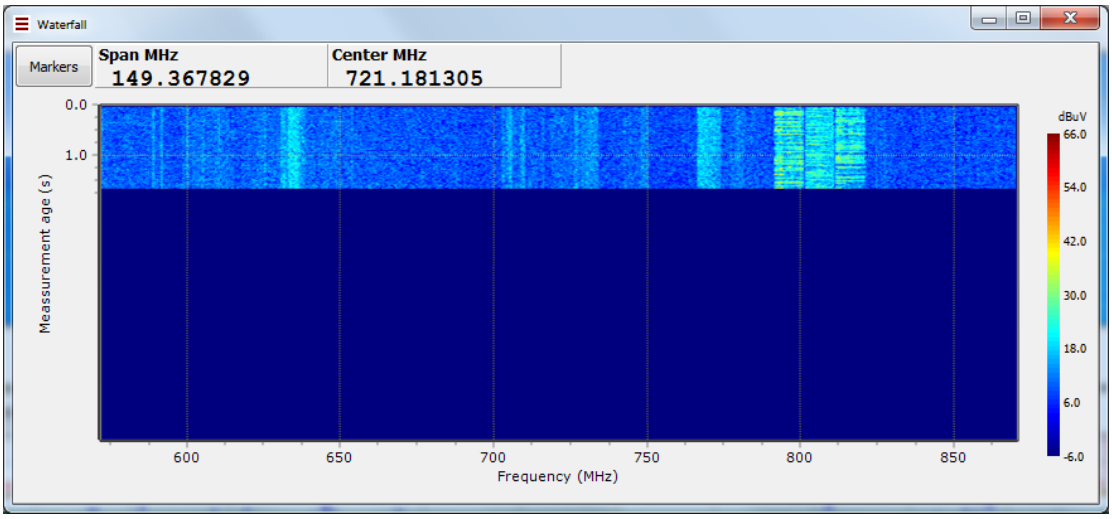
Třetí komponenta je `TChartZoom`. S její pomocí se otevře nové `TWaterfall` v režimu zoom, kdy jsou zobrazeny ovládací prvky pro snadné pohybování se po vodopádu a zaměření se na určitou část. Zároveň režim v okně dovoluje otevřít několik těchto oken zároveň pro sledování například na více monitorech, nebo v různé úrovni přiblížení.

Komponenta opět využívá série pro kreslení čáry. Ta slouží ke zvýraznění toho, která část vodopádu má být zobrazena v novém okně. Ovládání je opět řešeno pomocí myši. Protože levé tlačítko je použito k obyčejnému přibilžování, bylo nutné přijít s nějakou formou přepnutí módu, tak aby táhnutí nemělo za následek integrované přiblížení. Vytvoření tlačítka, nebo jiného grafického prvku, pro přepínání by vedlo k zesložitění grafického rozhraní. Zvolil jsem raději kombinaci klávesnice a kliknutí. Pokud tedy uživatel stiskne klávesu shift a následné klikne do vodopádu, dojde k ukotvení prvního bodu. Druhý pak sleduje umístění kurzoru a vykresluje se pod ním. Tím zvýrazňuje uživateli, který rozsah bude nově vytvořený vodopád zobrazovat(viz Obrázek 2.6). V tomto stavu je stále možné používat normální přibližování. Jakmile dojde k druhému kliknutí za současně stisknutého shiftu, přestane se kreslit pomocná čára a dojde k vytvoření nového okna vodopádu. To je iniciováno vytvořením události `ZOOMWATERFALL` v `EventQueue` a jejím následném zpracování v hlavním okně.

Ovládací prvky, které na běžném grafu nejsou, jsou `Span MHz` a `Center MHz`. Fungují tak, že `Center` nastavuje jaká frekvence má být uprostřed a `Span`, jak daleko od zvoleného středu má být vodopád ještě vykreslován. Tento způsob ovládání, namísto nastavování od-do jsem zvolil, protože se domnívám, že při sledování



Obrázek 2.6. Zvýraznění toho, nad kterou částí má okno vzniknout.



Obrázek 2.7. Vodopád ve vlastním okně v režimu zoom.

se uživatel zaměřuje na určitou frekvenci a její okolí a pro nastavení je tak přirozenější, nastavit frekvenci která ho zajímá.

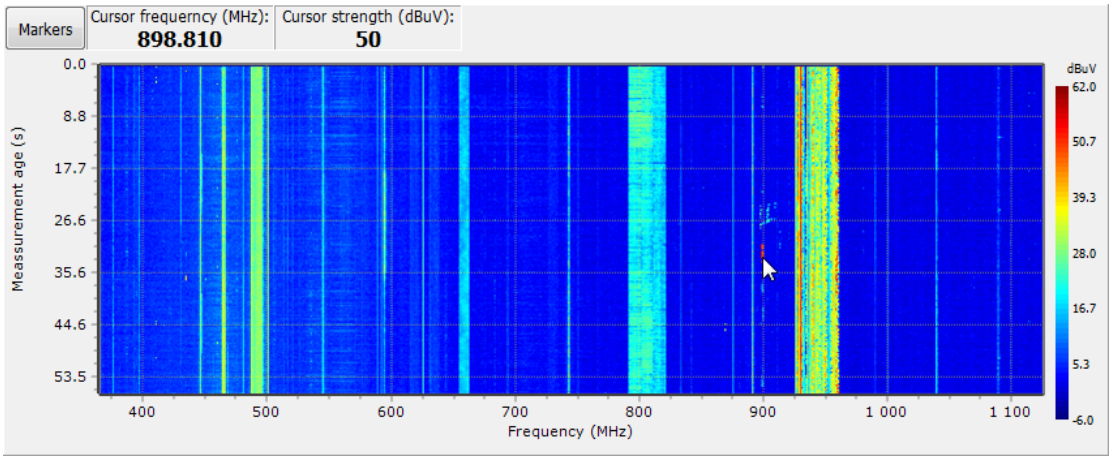
Rozdíl této formy přiblížení oproti standardní je v systémových nárocích. Při přiblížení na normálním grafu dochází ke zpracování celého řádku. Je udržováno mnoho řádků měření a to stojí desítky MB paměti. Ve vodopádu v režimu zoom je zpracována a uchovávána pouze část měření čímž dochází k úspoře paměti.

Protože je tento zoom vodopád pořád instance `TWaterfall`, fungují zde všechny věci jako v normálním režimu. Dá se použít přiblížení myší, měřit pomocí markerů, či vytvořit nový zoom vodopád.

Hned vedle kontrolních prvků vodopádu je zobrazena frekvence a úroveň signálu pod kurzorem. To je zde pro zrychlení vyčítání těchto hodnot, protože jinak se uživatel musí smířit s odhadem založeným na popiskách os, nebo využít přiblížení a z os to tak moci odečíst přesněji. Úroveň signálu pak šlo odhadnout pouze na základě barvy.

Hodnota frekvence se dala zjistit snadno. `TChart` disponuje callbackem `OnMouseMove()`, mezi jehož parametry je pozice kurzoru v oblasti grafu. Série grafu pak má funkci na převedení této hodnoty na X-ovou, resp. Y-ovou hodnotu os grafu. Tu pak už stačí jen vypsat. `TChartHandler` však tento callback používá. Je nutné jej proto přiřadit až po vytvoření instance.

Získat úroveň signálu byl poněkud větší oříšek. Pro vykreslování je použita paleta, jenž převádí osmibitovou hodnotu na RGB, která je následně použita k obarvení pixelů. Z důvodu optimalizace rychlosti zobrazení uchovávám kopii toho, co je zobrazeno na monitoru a převod pixelu na hodnotu, se tak redukuje na najití správné hodnoty v této kopii a odečtení 127.



Obrázek 2.8. Hodnota pod kurzorem.

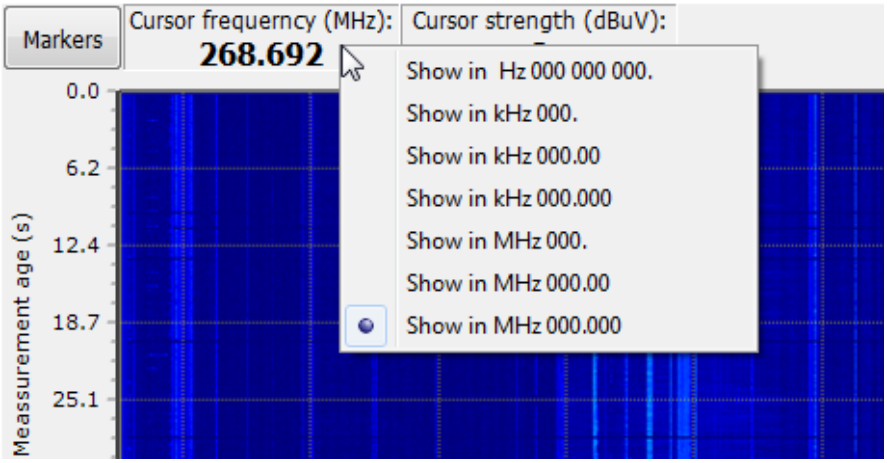
Záměrem bylo zjistit pozici levého horního rohu obrázku vodopádu, odečíst tyto hodnoty od hodnot, které jsem získal z callbacku a matematikou základní školy získat správnou pozici v kopii. Jednoduchá operace, avšak jak už to tak bývá, cesta k úspěchu bývá trnitá a ne vždy se věci povedou napoprvé. Nad touto drobností jsem strávil nepříjemně mnoho času.

Každý grafický objekt je určen čtyřmi parametry: výškou, šířkou a vzdálenostmi levého horního rohu od levého horního rohu rodiče ve kterém je. Vodorovné osy pak mají pouze vzdálenost od vrchu, vodorovné od levé strany. Kombinací těchto vzdáleností by pak neměl být problém zjistit požadovanou pozici. Vzdálenost od levé osy od strany byla uvedena správně. Vzdálenost horní osy (která není vidět) byla však nulová.

Jelikož jsem s horní osou neuspěl, zkusil jsem štěstí se spodní, která vidět je. Výpočet by tedy měl být pouze o malinko složitější o odečtení výšky obrázku pozadí. Po odečtení mi však při najetí kurzoru na vrchní řádek vycházela hodnota 14 pixelů místo nuly! Nevěda kudy kam jsem procházel editor nastavení grafu doufaje, že něco objevím. Nakonec jsem narazil na nastavení hodnoty margin - okraje. Ten byl v procentech. Nabízel možnost změnit jednotky na pixely a tak jsem nastavil hodnotu tak, aby graf vypadal zhruba stejně a zahrnul jsem tuto hodnotu do výpočtu. Nyní už byla pozice a tedy hodnota správně.

Rozhodl jsem se věc označit za hotovou a pročistit kód od ladících proměnných, když v tom mi to došlo. Při vykreslování grafu je výchozí nastavení hodnot na osách podle pixelových rozměrů pozadí. Osu x nastavuji na frekvenci, avšak osu y neměním a dokonce ji využívám ve výchozím stavu pro vykreslení vlastních popisků - času. Tím jsem se obloukem opět dostal k funkci série přepočítávající polohu kurzoru na hodnotu na ose.

Při zobrazení jsem váhal jaké použít jednotky. Nejmenší nastavitelný krok frekvence je 3 125 Hz, dává tak smysl zobrazovat jednotky Hz. Největší je pak 400 kHz, kde postačí zobrazovat desetiny MHz. Šel jsem tedy cestou kompromisu a vytvořil



Obrázek 2.9. Volba stylu zobrazení frekvence.

jsem kontextové menu (Obrázek 2.9) pomocí kterého si uživatel může zvolit styl zobrazení, jaký mu bude vyhovovat.

2.3 TLineGraph

Spektrum, podobně jako vodopád, byl součástí `DataProcesoru`. Po přepracování toho z původního návrhu příliš nezbylo. Data jednotlivých čar byla původně uchovávána ve struktuře `GraphLinesPointers`. Takovéto uchovávání rovněž nečinilo příliš problémů v případě hodnot odpovídající počtu pixelů na monitoru, při zvýšení na plné rozlišení měření však neustálá konverze `unsigned char -> integer` při zpracování a `integer - double` při překreslení zbytečně zabírá čas.

```
struct GraphLinesPointers{
    int Length;
    int * MaxHold;
    int * MinHold;
    int * Learned;
    int * Avg;
};
```

Graf byl obsluhován přes `GraphHandler` a data uchovávána a zpracována za pomoci `GraphLines`. `GraphHandler` pak provádí samotné překreslování v momentě zavolání funkce `repaint()`.

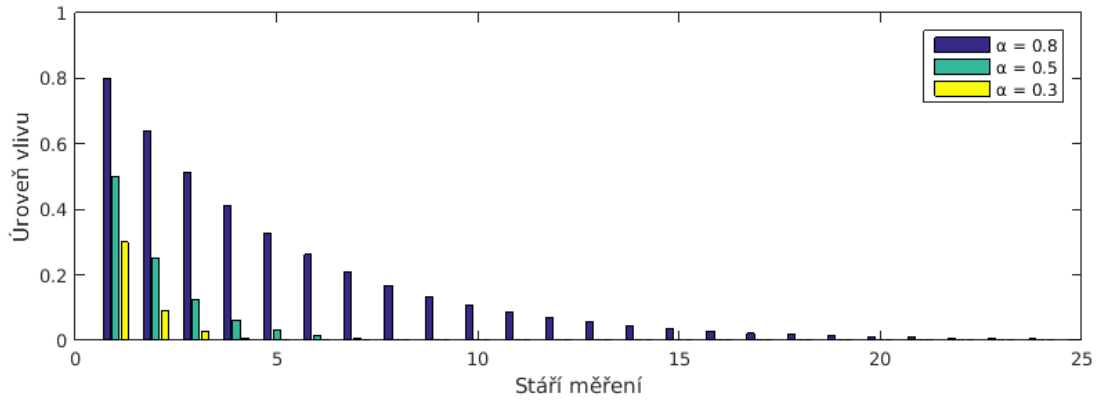
Aby sedělo provedení vykreslování spektra do koncepce zobrazovačů, vytvořil jsem dalšího potomka `TDisplay` a tím je `TLineGraph`. K němu patří `TLineGraphController`, který má za úkol zpracovávání a uchovávání dat. Ty jsou nově uchovávány jako `double`. Datová náročnost je sice dvojnásobná, ale v absolutních číslech jde pouze o 4 MB na jednu čáru - optimalizace paměti zde tak musí ustoupit optimalizaci výkonu.

K minholdu - reprezentující nejnižší dosaženou hodnotu, maxholdu - reprezentující nejvyšší hodnotu, aktuální hodnotě a té naučené přibyly dva nové. Exponenciální klouzavý průměr a celkový průměr. Celkový průměr je realizován neustálým sčítáním hodnot a následně jednoduchým podělením celkového počtu přijatých měření. Tím pádem může teoreticky dojít k přetečení a začnou se zobrazovat nesmyslné hodnoty.

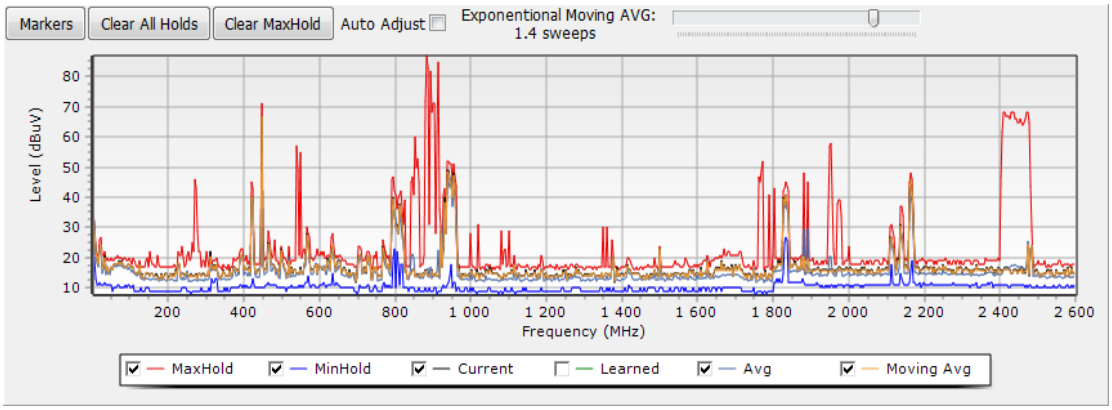
Hodnoty se uchovávájí do `double`. Jeho maximální hodnota je $1.7E308$. Maximální naměřená hodnota je nejvýše 127 - tedy ne více než $1E3$. Nejvýše je možné získat zhruba 1000 měření za sekundu - tedy opět $1E3$. Den má 86 400 sekund - to je řekněme $1E5$. Z toho vyplývá, že maximální hodnoty dosáhne součet odhadem za $1E297$ dní. To je $1E285$krát více, než je stáří vesmíru. Šance, že by tedy kdy došlo k přetečení je prakticky nulová.

$$S_1 = Y_1$$
$$pro\ t > 1,\ S_t = \alpha Y_t + (1-\alpha)S_{t-1} \tag{1}$$

Klouzavý exponenciální průměr je definován rovnicí (1). Průměr je inicializován prvním měřením S_1. Při každém dalším je přičten zlomek původních hodnot ke zlomku dosavadního průměru. Čím starší měření, tím menší má na velikost průměru vliv. Změnou hodnoty alpha v intervalu $< 0,1 >$ se ovlivňuje, jak velkou váhu mají mít jednotlivá starší měření. Na obrázku 2.10 je znázorněno jak se snižuje vliv jednotlivých starších měření vzhledem ke zvolenému α.



Obrázek 2.10. Vliv měření na průměr vzhledem k jeho stáří pro $\alpha = 0.8$.



Obrázek 2.11. Spektrum.

Ovládání relevantní pro spektrum jsem změnou návrhu přesunul z horního panelu přímo ke grafu a přibylo možnost využití markerů k měření zobrazených hodnot. Tlačítka `Clear All Holds` a `Clear MaxHold` slouží k vymazání všech

akumulujících čar, resp. jen maxholdu. Jelikož je odděleno zpracování a zobrazení dat, je pouze nastaven příznak, že má k vymazání dojít a dojde k němu při přidání dalšího vzorku. Zaškrtávátko `Auto Adjust` slouží k aktivaci automatického nastavování rozsahu levé osy tak, aby se přizpůsobila zobrazoaným hodnotám. Posuvníkem se pak nastavuje hodnota α pro exponenciální průměrování.

α je nicneříkající číslo a tak, pro výpočet z kolika vzorků bude průměr počítán využívám vzorce $N = (2/\alpha) - 1$. Ten je odvozen z výpočtu N-denního exponenciálního pohyblivého průměru[2]. Výpočet je takto nastaven pro to, aby N vzorků mělo v součtu váhu 0.8647.

Pomocí legendy jde zobrazovat, či skrývat jednotlivé průběhy. Nastavení jejich zobrazení je spolu s hodnotou α ukládáno pomocí `TSettings` do ini souboru a vydrží tak do dalšího spuštění programu.

Jak jsem uvedl na začátku kapitoly. Data jsou uchovávána v polích typu `double`. Pro každou sérii jednotlivě. Ty jsou aktualizovány s každým příchozím měřením podle toho, jakého je typu. Před samotným přidáním však může dojít k vynulování některých, pokud je nastaven příslušný příznak.

Ekvivalentně se zobrazením dat ve vodopádu i zde se naráží na problém příliš mnoha bodů (vykreslení trvalo příliš dlouho a posun po grafu měl delší odezvu) a tak bylo i zde nutné přistoupit ke kompresi před zobrazením. Rozdílná povaha však nedovoluje použít na všechny metodu hledání maximální hodnoty.

Tou jedinou výjimkou je MinHold. Slouží k zobrazování minimální hodnoty a tak při kompresi těchto hodnot hledám vybírám vždy tu nejnižší hodnotu.

2.4 TTimeView

Zobrazení v čase je nový zobrazovač, který je podobný osciloskopu. Stejně jako předchozí dva je i tento potomkem `TDisplay`. Jeho cílem je zobrazit časový průběh na zvolené frekvenci. Ta je volena uživatelem. Průběh je v čase a ten je na spodní ose zobrazen jako skutečný čas naměření dané hodnoty. Na svislé je úroveň naměřené hodnoty.

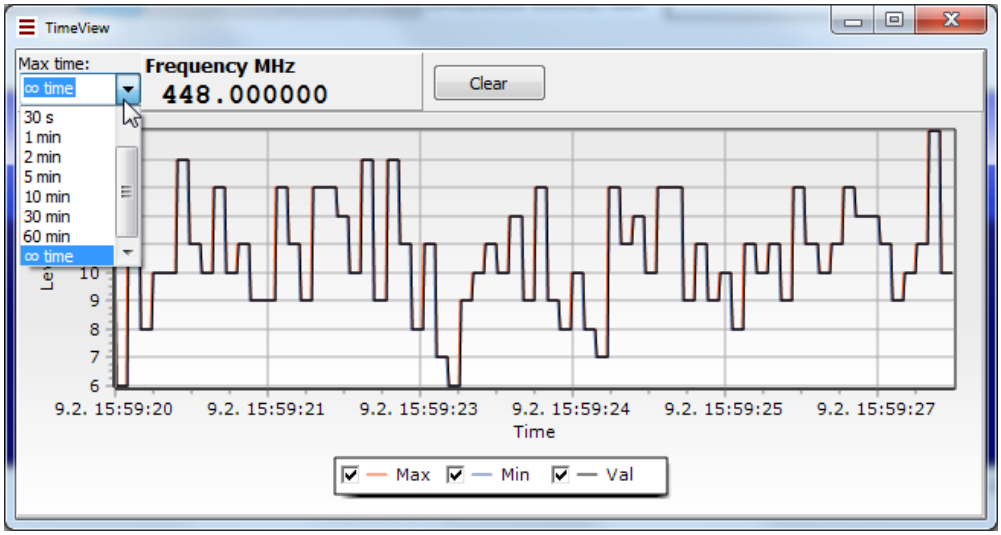
Uživateli je rovněž dána možnost zvolit si omezení času zobrazení skrz roletkové menu jak je vidět na obrázku 2.12. Uživateli může volit mezi časy začínající na 10 s a končící na 60 minutách. Poslední možností je nekonečný čas.

V zobrazení jsou tři průběhy označených - Max, Min a Val. Rozdělení na tyto 3 průběhy souvisí s kompresí. Při málo bodech - tak, jak je to k vidění na obrázku 2.12 - kdy je počet zobrazených hodnot menší, než šířka v pixelech, na kterém je graf zobrazen, dochází k natažení grafu. Jedna hodnota je tak zobrazena přes více pixelů a graf je bez problémů čitelný.

V opačném případě, kdy je bodů více, by docházelo při vykreslení všech bodů, kromě zpomalení odezvy, ke slití jednotlivých svislých čar grafu a tím k vyplnění tvaru vymezeného nejvyšší a nejnižší hodnotou dosaženou v měřeních připadající na příslušný pixel monitoru.

Rozhodl jsem se proto kvůli zvýšení přehlednosti při zobrazení rozdělit průběhy těchto tří průběhů. Min a max znázorňuje nejnižší resp. nejvyšší naměřenou hodnout připadající na pixel. Účelem hodnoty označené v legendě grafu jako Val by mělo být uživateli přiblížit průběh mezi nejvyšší a nejnižší hodnotou.

Nabízí se zde celá plejáda možností, jak si s reprezentováním několika hodnot jednou poradit. Různé druhy průměrů, medián, modus,... Na rozdíl od šířky mě-

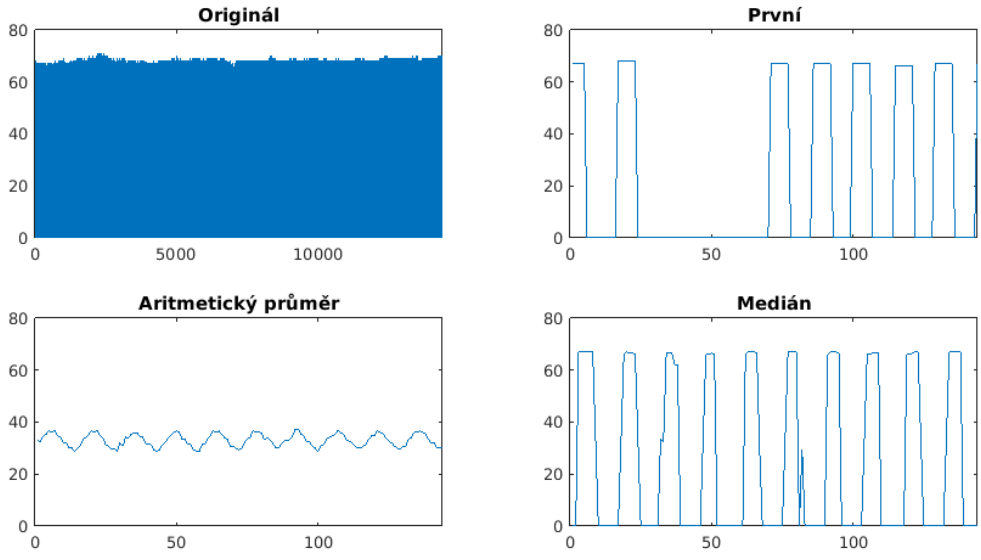


Obrázek 2.12. TTimeView ve výchozím zobrazení.

ření, které je omezeno na milion bodů, pří použití intervalu nekonečno, zde toto omezení neplatí. Při omezení na kratší interval i tak může docházet k nahromadění velkého počtu bodů při rychlém měření - rychlost zpracování je tak podstatná. S ohledem na to jsem zkusil porovnat ty, které nevyžadují příliš mnoho procházení, nebo počítání.

Pro volbu nejvhodnějšího jsem se rozhodl nasimulovat ty nejslibnější - aritmetický průměr, medián, modus a výběr první hodnoty(když několik hodnot připadá na jeden pixel, vezmu první hodnotu).

Pro ověření jsem vzal dva signály, které jsem naměřil. První je periodický - víceméně jde o čtvercový signál. Druhý je těžko popsatelný - ničím nevyniká, jde o šum s nepravidelnými výkonovými špičkami.



Obrázek 2.13. Snížení počtu vzorků - decimace.

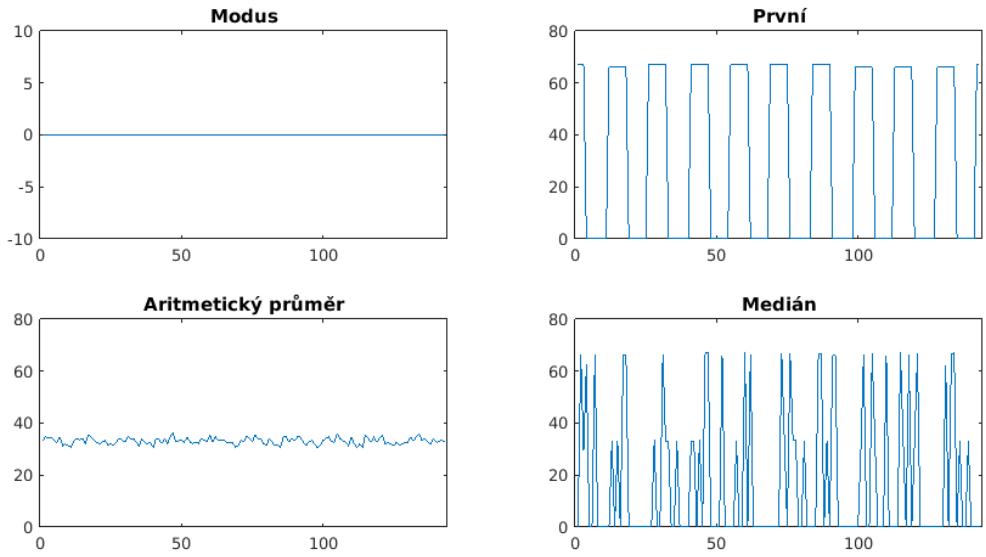
Signál je tvořen 14400 bod, které jsou zredukovány 100:1, tedy na 144 bodů. Na obrázku 2.13 je vidět, jak vypadá originální signál a proč bylo vhodné k redukci přistoupit - je vyplněn celý prostor mezi nejvyšší a nejnižší hodnotou. Pro porovnání účinku jsem redukci provedl ještě jednou(obrázek 2.14) a to pro signál, který je posunutý o 30 bodů.

Modus, je víceméně nepoužitelný. Na obrázku 2.14 je vidět, že nejčastější hodnota je nula. To je údaj sice pravdivý, ale vlastním průběhu toho příliš neříká.

Vybrání prvního dává v druhém případě velmi přesný obrázek toho, jak signál vypadá. Bohužel, na obrázku prvním dochází k nepříjemnému efektu. Perioda vybíraného bodu je shodná, nebo velmi podobná periodě signálu a dochází tak k úplnému zastření toho, že tam nějaký signál je. Spoléhat tedy na náhodu, že se takto periody nepotkají a že jeden, ve své podstatě náhodný, vzorek rozumně reprezentuje zbytek skupiny, nelze.

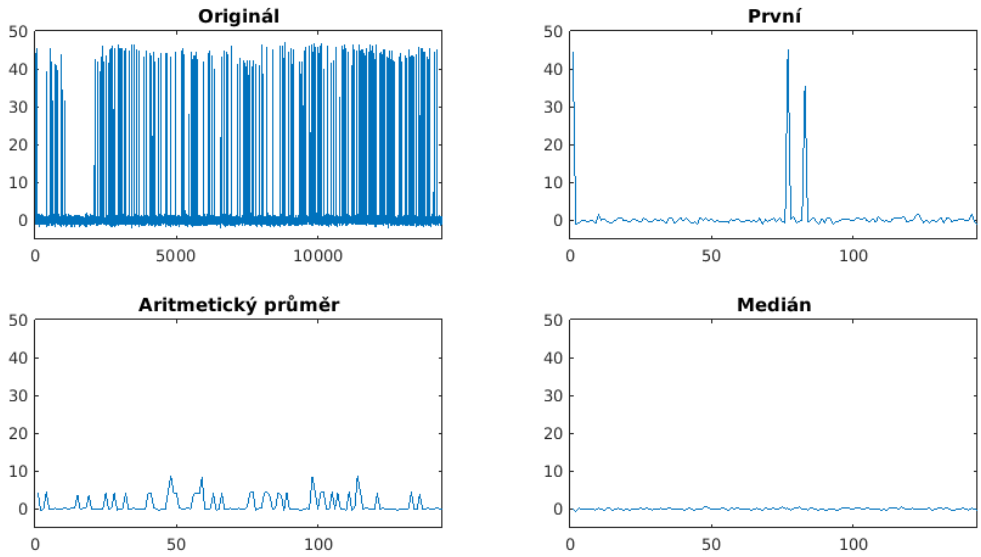
Medián v prvním případě poměrně dost vypovídá o průběhu signálu. Reprezentuje větší část vzorků, ukazuje aproximaci skutečného průběhu a dává tak pěknou představu. V druhém (posunutém) případě už to tak přesné není. Místo pěkných periodických signálů se objevují jednotlivé signálové špičky, které působí poměrně náhodným dojmem.

Aritmetický průměr v sobě nese informaci o všech bodech, ze kterých je tvořen. V prvním vzorku tvoří opakující se signál a dává tak tušit, že jde o periodický signál. V druhém však průměr žádný vzor netvoří. Přesto však dává tušit, že zde signál o nějaké úrovni je.



Obrázek 2.14. Snížení vzorků posunutých o 30 měření.

Tento vzorek je tvořen převážně šumem a víceméně náhodnými a nepravidelně se opakujícími špičkami. Metoda první hodnoty překvapivě opět ukazuje poměrně přesně jak signál vypadá - šum a občas nějaká špička. Medián i modus (ten není znázorněn) zde kompletně pohořel. Ukazuje pouze nejčastěji se vyskytující hodnotu šumu, zatímco špičky se vůbec nevyskytují. Aritmetický průměr je pak dostatečně věrný obraz průběhu.



Obrázek 2.15. Snížení druhého vzorku dat.

Z pokusů vyplývá, že modus je úplně nepoužitelný. Zobrazení mediánem závisí na tom, jaká jsou data. Může avšak nemusí zobrazovat informaci o tom, jak signál vypadá. Překvapivě dobře dopadlo vybírání prvního vzorku věrně ilustrující, jak signál vypadá. Aritmetický průměr ukazuje, jak je to s úrovní signálů a intuitivně tak dává největší smysl.

Nejlepší volbou se tak jeví aritmetický průměr. Průběh signálu je shora a spoda vymezen největší a nejnižší hodnotou. Průměr pak ukazuje k jaké hodnotě se body přibližují a pro signál mající občas nějakou větší hodnotu a jinak šum ukazuje, zda se někde nevyskytuje větší koncentrace nešumových hodnot a tedy místo hodné bližšího zkoumání. Druhou nejlepší a je vybírání prvního bodu. Je však závislá na tom, aby signál byl periodický a celkově sednul pěkně. Vytváří však zavádějící dojem o celkovém průběhu. Není jisté, zda signál na obrázku 2.14, který pěkně vychází, je periodický s malou periodou, nebo signál s velkou periodou - pro velkou nepředvídatelnost jsem se rozhodl tuto metodu nepoužít.

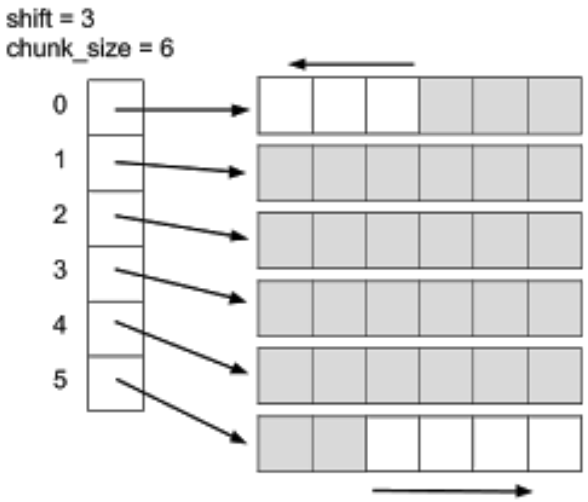
Pro realizaci osciloskopu s pevnou vzorkovací frekvencí je jednoznačná volba použít kruhový buffer. Tento však pevnou délku nemá. To je způsobeno tím, že nelze přesně předvídat jak rychle a kolik vzorků je na daný časový úsek potřeba. V případě nekonečného časového okna pak nelze ani vytvořit jakýkoliv odhad, protože jeho délka neustále narůstá.

Pooužil jsem proto `std::deque`. Je to poli podobná struktura, podobně jako `std::vector`. Na rozdíl od vectoru však umožňuje operace `push` a `pop` na začátek i konec s těmito složitostmi:

- O(1) pro `push_back()`, `push_front()`, `pop_back()`, `pop_front()`
- O(1) pro náhodný přístuo skrze `operator[]`
- maximálně N/2 posunů při operaci insert/remove, pro deque velikosti N

Těchto složitostí je dosaženo tím, že deque je implementováno tak, že je jedno pole, držící ukazatele na další pole konstantní velikosti, do kterých jsou prvky ukládány (viz obrázek 2.16). U prvního a posledního takového pole je uložena pozice prvního, resp. posledního, prvku. Při přidání prvku pak buď dojde pouze k

posunutí pozice, nebo v případě, že dojde k naplnění příslušného pole, k alokování nového a vložení prvku. Při takovéto alokaci může dojít k tomu, že se naplní pole, udržující ukazatele na jednotlivá pole s prvky. Musí tak dojít k jeho realokaci. To je relativně náročná operace, avšak pokud je zvolena dostatečná velikost polí s daty, nehraje toto zdržení velkou roli, jelikož čas této operace bude rozpuštěn v ostatních operacích.



Obrázek 2.16. Znázornění[3] implementace std::deque[3].

Data do `TTimeView` přicházejí skrze `addSweep()`. Následně je na základě počáteční frekvenci a velikosti kroku trojčlenkou dopočítáno, který pixel je požadován uživatelem podle nastavené frekvence. Jak pixel, tak čas, kdy bylo měření přijato je uloženo do deque.

Při zobrazení takto uložených dat jsem musel řešit nečekaný problém. Komponenta TeeChart, kterou používám pro všechny, používá pro nastavení os datový typ `double`. Stejně tak je v `double` uchováván časový okamžik v `TDateTime`.

Jeden den zde odpovídá hodnotě 1. Počet dní se počítá od času 00:00 30.12.1899. (ano opravdu od 30.12.1899 a ne 1.1.1990). Pro dnešní rok 2016 to tedy odpovídá zhruba 42 000 dní. Při této hodnotě je nejmenší možný inkrement 7e-12. Tato hodnota odpovídá v `TDateTime` 0.6286 μs, což není zase tak malá hodnota, jak by se zdálo. Při rychlém skenování jsou měření přijímána až každých 200 μs.

`TTimeView` může teoreticky začít vykreslovat graf už při dvou vzorcích. Řekněme, že je rozdíl v čase těchto dvou vzorků právě těch 200 μs a šířka grafu je 1000 px. Každý pixel bude v tomto případě reprezentovat 0.2 μs, což je méně než možné minimum 0.6286 μs. 1000 px je ještě navíc v době začínajících 4k monitorů a běžných FullHD poměrně střízlivý odhad. Navíc ani nevím, jak je uvnitř `TChart`u implementováno počítání os. Může dojít z nějakého důvodu k násobení těchto čísel a tím ještě většímu snížení rozlišení.

Z toho důvodu zobrazení takovýchto dat prostě není možné.

Ven z této prekérní situace pomohl trik s takzvanou `Bulharskou konstantou`. Pro reprezentování hodnoty je potřeba jak čas jeho přijetí, tak jeho hodnota.

Při přijmutí prvního vzorku si uložím čas jeho přijetí, ne však do deque, avšak do zvláštní proměné. Do čas uložím až po tom, co od něho tento čas odečtu a vynásobím onou Bulharskou konstantou, jejíž hodnota je 100 000. Tuto operace (odečtení referenčního času a vynásobení konstantou) provedu při každém uložení nového vzorku. Úroveň signálu uložím tak, jak je. S hodnotami na úrovni 7e-7 si už graf poradí.

Protože na vodorovné ose jsou nyní řekněme nesmyslné, bylo nutné přistoupit k vlastnímu generování popisků. Jelikož stejně při vykreslení počítám hodnotu pro každý pixel není to problém. Každých 100 pixelů při vložení jednoho popisku dopočítám skutečný čas přijetí, vydělením konstantou a přičtením referenčního času.

Na rozdíl od přepočítávání vodopádu nebo spektra, nelze předem spočítat, kam který bod na grafu připadne. Jediné co dopředu spočítat jde, je který čas případne na který bod, protože první a poslední hodnota přepočítaného času je známa a navíc vždy začíná nulou. Paradoxně je tak toto přepočítávání o něco snazší. Principielně, jeden while cyklus slouží k iteraci přes jednotlivé pixely vnořeným pak iteruji přes naměřené hodnoty, které jsem ještě nepoužil a jsou přijaty před časem následujícího pixelu.

Zde je řešeno i počítání hodnot min, max a průměru. Průměr je počítán akumulátorem, který je na konci vydělen počtem hodnot kterými je tvořen.

Při aplikaci časového omezení je kontrolováno, zda rozdíl přijetí mezi nejnovějim a nejstarším měřením není více než nastavené maximum. V takovám případě dojde k odečtení času druhého nejstaršího měření od všech ostatních, přičtení (po správné úpravě) k referenčnímu času a vymazání nejstaršího měření pomocí `pop_front()`. Opět zkontroluji zda je rozdíl dostatečný a případně postup zopakuji.

2.5 Ostatní

2.5.1 EventQueue

`EventQueue` je thread-safe fronta sloužící k předávání informací a řízení hlavního grafického okna. Princip jejího fungování je takový, že si nějaký objekt vytvoří zprávu naplnením informací do struktury a přidá ji do této fronty. V hlavním okně se pak kontroluje zda ve frontě není nějaká zpráva. Pokud fronta nějakou obsahuje, vyjme ji a interpretuje.

Akce, kterou následně vykoná, může být od přepsání textu, až po zobrazení nového okna. Pro hlavní rozlišení, kterou akci vykonat slouží výčtový datový typ. Oproti verzi z bakalářskén práce se počet akcí více než zdvojnásobil, aby byly pokryty všechny nové schopnosti programu.

```
enum EventType {CONNECTIONSTATUS, PARAMETERS, PACKETCOUNT, DEVICECHOSEN,
READTEMPERATURE,MOUSECOLORSVALUE, SAVEIPSETTINGS, SAVELISTENINGSETTINGS,
MHZRECEIVED,CMDTRANSFERRED, PLAYBACKTIME, PLAYBACKFILEOPENFAILED,
PLAYBACKEOF, RECORDINFO, DELETEDISPLAY, ZOOMWATERFALL,
SHOWREBOOTINFOWINDOW, MWANTSTATUS};
```

Původní struktura obsahovala pouze typ události celočíselnou hodnotu. Z důvodu rozšiřování funkcionality bylo nutné rozšířit strukturu o předávání dalších informací - přibyl `float`, `String` a ukazatel `void *`.

```
struct Event{
    EventType Type;
    int IntegerVal;
    float FloatVal;
    String StringVal;
    void * Ptr;
};
```

2.5.2 TSettings

Pro ukládání konfigurace programu byl použit ini soubor. Nabízí přehledně strukturovaný způsob ukládání proměnných a dovoluje tak snadnou editaci nastavení bez nutnosti spouštět program, nebo snadnou záměnu konfigurací vyměněním souboru za jiný.

Pro přístup ke konfiguračnímu souboru je v Embarcadero Studio dostupná třída `TIniFile`. Ta poskytuje rozhraní pro přístup ke konfiguračnímu souboru a nastavení, neboť přečtení nějaké hodnoty je tedy otázka jednoho volání funkce. To je ideální pokud chci pracovat s konfiguračním souborem z jednoho místa.

Problémy nastávají, pokud chci k souboru přistupovat z více míst a více vláken, což je můj případ. Prvním problém by mohl nastat, pokud bych se z nějakého důvodu rozhodl přestat používat název souboru `settings.ini`, nebo dovolit uživateli vložit název konfiguračního souboru jako parametr při spuštění. První případ je zdlouhavý a nepohodlný, ale řešitelný - musel bych projít celý zdrojový kód a změnit každý jednotlivý výskyt. Řešení druhého problému by pak zahrnovalo dopsání funkcí pro nastavení názvu souboru do každé třídy a zajistit, že budou všechny správně zavolány.

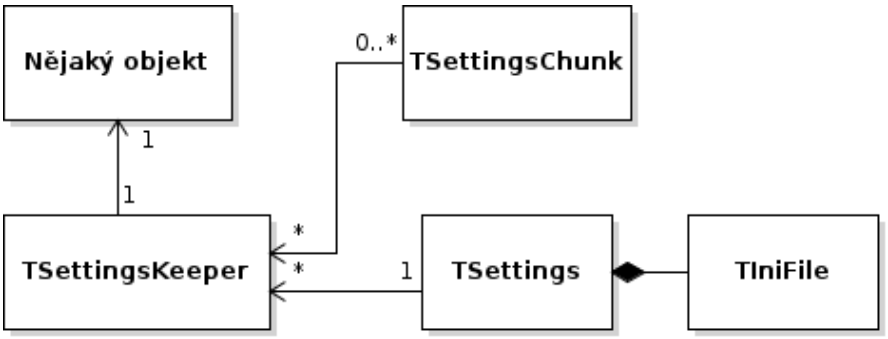
Dalším problémem, který může být ale nemusí, je ten, že se mi nepodařilo spolehlivě zjistit, zda je `TIniFile` thread safe. A jelikož moje aplikace vlákna nezřídka používá, mohly by se začít objevovat nedeterministické problémy - a ty se velmi těžko hledají.

Abych těmto a podobným obtížím a problémům předešel, rozhodl jsem se proto raději místo přímého používání `TIniFile` vytvořit adaptér pro jeho používání:

```
class TSettingsKeeper{
private:
std::vector<TSettingsChunk> FSettingsChunks;
...
public:
    TSettingsKeeper();
    ~TSettingsKeeper();

    void SaveSettingsToFile();
    void LoadSettingsFromFile();

    void SetInteger(String Category, String Name, int Value);
    void SetBoolean(String Category, String Name, bool Value);
    void SetDouble(String Category, String Name, double Value);
    void SetString(String Category, String Name, String Value);

    int GetInteger(String Category, String Name, int DefaultValue);
    bool GetBoolean(String Category, String Name, bool DefaultValue);
    double GetDouble(String Category, String Name, double DefaultValue);
```

```
    String GetString(String Category, String Name, String DefaultValue);
};
```

Kromě zpřístupnění možnosti ukládání do souboru slouží adaptér zároveň jako vyrovnávací paměť pro uchovávání nastavení, před samotným zápisem do souboru, aby se minimalizovalo množství jednotlivých přístupů na pevný disk.



Obrázek 2.17. Diagram použití TSettingsKeeper.

Princip použití je jednoduchý. Znázorněn je na Obrázku2.17. Objekt, mající zájem využít ukládání nastavení do souboru si vytvoří instanci `FSettingsKeeper`. Zavoláním `Set___()` inicializují `TSettingsChunky`, které slouží jako informace, které položky konfiguračního souboru daný objekt požaduje. Hodnota, která se zde nastaví slouží jako defaultní hodnota, pokud tato položka není v souboru obsažena.

V `TSettingsChunk` je uložena hodnota a její typ, kategorie a jméno položky položky ini souboru.

Zavoláním `LoadSettingsFromFile()` se získá singleton instance `TSettings`, která disponuje thread safe metodami, dávkově přistupující k jediné instanci `TIniFile`, provádějící samotný zápis, resp. čtení ze souboru. Vláknová bezpečnost je zajištěna pomocí mutexu `TCriticalSection` k jehož zamknutí dojde okamžitě po vstupu do příslušné funkce - k tělu této funkce a tím i zápisování, se tak může dostat pouze jedno vlákno. Tento zámek je společný jak pro načítání, tak pro ukládání do ini souboru. Může tak skutečně probíhat jen jedna IO operace.

Pomocí funkcí `Get___()` lze po načtení ze souboru získat hodnoty, jenž byly načteny. Zde je defaultní hodnota použita v případě, kdy nebyl připraven příslušný chunk.

Pro uložení do souboru je opět nutné nastavit jednotlivé položky funkcí `Set___()`. Hodnota třetího parametru - `Value` - je pak ta, která bude uložena. Po připravení všech položek stačí zavolat `SaveSettingsToFile()` a dojde k bezpečnému uložení konfigurace na disk.

Kapitola 3
Nové vlastnosti

3.1 Nahrávání do souboru, jeho přehrávání a timeshift

3.1.1 Nahrávání

Novou schopností programu je nahrávání do souboru. To je užitečné pro zpětnou analýzu dat. Není nutné u měření sedět a sledovat celý průběh.

Při ukládání je nejdůležitější si nejprve rozmyslet jak s daty budeme pracovat a jaké množství dat bude program generovat. Data chceme samozřejmě být schopni přehrát a zobrazit je tak, jako kdybychom se dívali na živý stream z přístroje a určitě je budeme chtít nějak analyzovat. Co se velikosti týče, příjem dat je až 10 MB/s, což činí cca 30 GB za hodinu.

Začnu velikostí. Kvůli detekci je potřeba počítat s dlouhodobým nahráváním, které se nezřídka může pohybovat v řádu dnů. S maximálním rozlišením tedy není problém dostat se na TeraByty dat. Je tedy nutné vyřešit nějakou formu zmenšení ukládaných dat, aby nebylo nutné používat disková pole. Další problém s velikostí je rychlost načítání. Běžný desktopový disk má rychlost čtení cca 150 MB/s. Jednoduchým výpočtem 30 GB * 24 hodin / 150 MB/s zjistíme, že samotné načtení dat za jeden den bude trvat v ideálním případě 80 minut.

Samotné uchovávání a práce se soubory této velikosti je problematická. Microsoft Windows - platforma, pro kterou je program vyvíjen - pracuje nejčastěji se systémem souborů NTFS (pevné disky) a FAT32 (USB flash disky). Maximální velikost souboru pro FAT32 je 4 GB[4] a pro NTFS je to 16 EB[4], což je 16 milionů TeraBytů. Z těchto dvou limitů je podstatný ten menší, který je z hlediska aplikace opravdu malý.

Řešení problému maximální velikosti omezením maximálního nahrávacího času odvozeného od parametrů měření není vhodné. Vyžaduje totiž neustálé dozírání na probíhající měření lidskou obsluhou a znovuspuštění s ukládáním do nového souboru. Řešením by tedy mohlo být celý proces nějak zautomatizovat, aby výsledkem bylo jedno měření rozdělené do více souborů ideálně tak, aby tímto rozdělováním nebyl uživatel zatěžován.

Aby přehrávání bylo co nejvíc ekvivalentní s prací se samotným přijímačem je potřeba uložit stejné informace, jako jsou k dispozici když s ním pracuji. Tedy konfigurace měření a čas přijetí daného řádku měžení. Tato data jsou příhodně uložená ve struktuře 2.1, která je již používána a bylo by tedy šikovné ji použít znovu.

Při volbě ukládání dat je vždy několik možností. Můžou se ukládat jako do textu. Z těch lze jmenovat standardy XML, JSON případně YAML. Všechny tyto textové formáty jsou vhodné pro ukládání strukturovaných dat, kombinaci textů a čísel, jsou vymyšlené pro snadnou čitelnost a editovatelnost člověkem a strojem. Zde však jde o ukládání mnoha a mnoha čísel. Není nutné ani vhodné data

přímo upravovat přepisem hodnot v souboru a vzhledem k jejich množství je nepravděpodobné, že by uživatel toužil po tom, jednotlivé hodnoty číst jednu po druhé.

Alernativou je databáze. Nabízí možnost snadné práce se strukturovanými daty. Zvládá ukládat čísla i binární data. Jeden řádek tabulky by mohl představovat jedno měření. Výhodou by mohla být velká škálovatelnost při opravdu dlouhodobém měření, kdy by se program připojil k nějakému serveru s velkým diskovým polem. Toto řešení však vyžaduje zvláštní program, místo, kde databáze poběží. Navíc není možné přenášet jinak než buď překopírováním do jiné databáze online, nebo exportem databáze do souboru a jeho následným načtením v jiném počítači.

Rozumnou volbou se tedy zdá ukládat data binárně - budou čtena jen programem a výsledkem je datový soubor, který zkopírujeme na flashdisk a snadno přeneseme. Pro binární ukládání naměřených hodnot se přímo nabízí formát TDMS od National Instruments[5]. Ten je v porovnání se textovými strukturovanými daty menší. Vyžaduje však externí knihovnu a je zbytečně komplexní pro účely aplikace.

Logickým důsledkem je tedy proprietární formát. Ten musí obsahovat dostatek informací pro přehrání, nemít zbytečně komplikovanou strukturu a dovolovat snadnou přenositelnost naměřených dat.

V jednom z přechozích odstavců jsem nastínil potřebu komprese. Ta je vhodná z důvodu velkého konstantního datového toku atakující 700 GB denně. Ta by měla být v podstatě bezztrátová - záměrem je přeci detekovat signály, tedy ty, které tam nebyly a nyní jsou. To věc velmi zjednodušuje.

Přístroj je schopen celé své spektrum projít za jednotky až malé desítky milisekund. Takto jemné časové rozlišení při měření ve výsledku není nezbytně nutné. Důležité je, zda tam nějaký signál vůbec byl. Snížit rychlost skenování umělým nastavením nižší vzorkovací frekvence pomocí zabudované možnosti prodlevy mezi jednotlivými vzorky, a uměle tak degradovat schopnosti přijímače, není vhodné. Některé signály trvající kratší dobu by mohly uniknout odhalení.

Nejjednodušší a nejsnadnější se tedy zdá vzít několik měření s nulovou prodlevou a najít mezi nimi pro každou frekvenci nejvyšší naměřenou hodnotu. To by mohlo při nastavení vzorkování po stovkách milisekund až jednotkách sekund snížit datový tok na méně jak desetinu, což by mělo být dostatečné pro udržení množství dat na uzdě, při zachování kvality měření.

Vybíráním nejvyšší hodnoty z několika měření je dosaženo značného zmenšení výsledných souborů, ale ani s touto technikou ještě není možné udržet velikost souboru na méně než 4 GB při dlouhodobých měřeních. Implementace ještě větší komprese příliš nepomůže. Při experimentální kompresi jednoho gigabajtu programem WinRAR nastaveným na největší kompresi byl výsledkem soubor o velikosti 500 MB. Zmenšení tedy bylo pouze poloviční. Komprese nepomůže, nezbývá tedy než dělit nahrávaný soubor na menší části.

3.1.2 Datová struktura souboru

Už mám rozmyšleno, že soubor bude komprimován jednoduchým algoritmem a rozdělen na více částí tak, aby bylo možné je přenášet na flashdisku s FAT32. Nyní se podívám na samotnou strukturu souborů.

Při otevírání souborů bych si měl být jistý, že je otevírán soubor patřící k mé aplikaci. Pro tyto účely se používá tzv Magic number[6]. To je řetězec bajtů na

začátku souboru, nebo přenosu dat sloužící k rozlišení souboru. Hlavní myšlenkou je, že je velmi malá pravděpodobnost, aby nějaký soubor začínal stejně. Zvolil jsem řetězec "`IP-982 `" - označení programu. Dvě mezery jsou přidány z důvodu zarovnání na celých 32bitů.

Kromě komprese použiji při nahrávání dělení na více jednotlivých souborů a stejně jako při ověření správnosti souboru jako takového, je vhodné ověřit i to, že jednotlivé soubory patří k sobě. K tomu se využiji stejný princip, jako je u Magic number - pravděpodobnost. Pro každé měření vygeneruji náhodné 64 b číslo.

Když už je prakticky jisté, že jsou k přehrávání vybrány správné a navazující soubory, co když bude potřeba časem udělat úpravy? Pro jistotu ještě přidám číslo označující verzi souboru.

Dále bude vhodné přidat informace o samotném měření. Pro rychlý přehled při načítání přibude čas prvního záznamu a jeho konfigurace.

Výsledkem je struktura (Tabulka3.1) obsahující všechny potřebné informace.

Typ	Název pole	Popis
uint8_t[8]	Magic	Kontrolní řetězec typu souboru
uint64_t	ID	Kontrolní řetězec měření
uint32_t	Version	Verze
double	BeginTime	Vzdálenost naměřených bodů
Struktura z tabulky 2.1	Conf	Struktura hlavičky měření

Tabulka 3.1. Hlavička souboru měření

Hlavička je vymyšlená, ale je nutné ještě uložit samotná data. Záměrem přehrávání tak, jako by se jednalo o přímou práci s přijímačem. Data z přijímače jsou kompletována a předána zobrazovačům tak, že obsahují konfiguraci přijímače a data. Protože ona struktura obsahuje vše potřebné - čas, délku, pozici ve spektru, atd.. Není důvod se tohoto schématu nedržet i nyní. Hned za hlavičkou souboru tak bude první měření ve formátu konfigurační hlavička dat, data, konfigurační hlavička dat, data,... dokud se neukončí měření, nebo nepřesáhneme limitní velikost jednoho souboru.

Hlavička prvního je daná, co ale s dalšími navazujícími soubory? Není důvod nepoužít stejnou hlavičku souboru a je jedině logické pokračovat ve vzorci střídání konfigurační hlavičky a samotných dat.

Dobrá tedy, otevřeme soubor, začneme jej přehrávat, dojdeme na konec a co pak? Který soubor ve složce je ten správný? Zde si pomohu příponou souboru. První souboru se bude jmenovat `nazev.cap`, další pak `nazev.cap1`, další `nazev.cap2`, atd. Tímto schématem vznikne snadno rozlišitelné pořadí souborů.

Vedlejší efekt mnou zvoleného způsobu ukládání dat je bonus ve formě možnosti otevřít si a přehrát kterýkoliv soubor samostatně. Obsahuje vše potřebné - od konfigurace přes samotná data a nic tomu tak nebrání.

3.1.3 Přehrávání

Soubory už uložené jsou. Přehrávání je poměrně přímočaré. Je nutné soubor otevřít, rozparsovat a správně interpretovat.

Pokud se podívám na stávající tok jednotlivých měření (Obrázek 2.3) a budu se držet záměru aby se s daty pracovalo jako se zařízením, je na pohled jasné,

že data musejí projít třídou `TDisplayManager`. Díky tomuto hrdlu je nasnadě, že data musejí přicházet odtud.

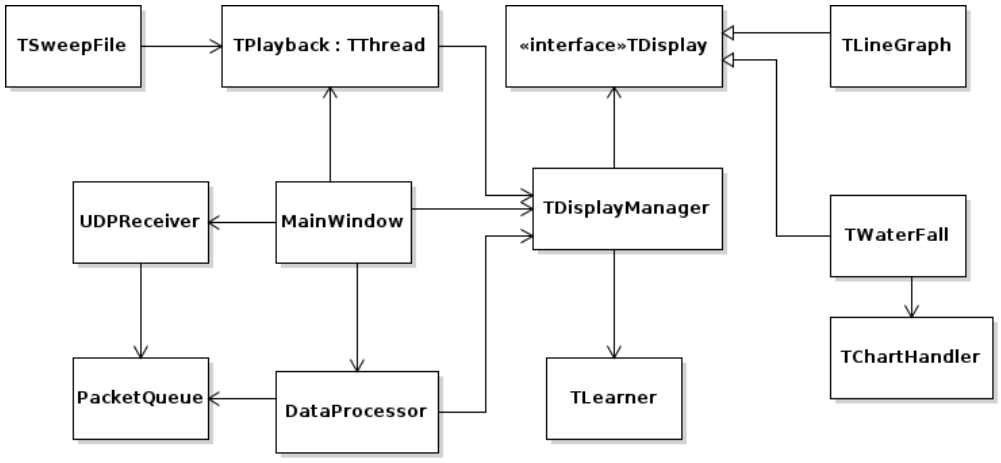
Při přehrávání je trochu jiná situace, než při zobrazování živých dat. Data už jsou uložena a uživatel nikam nespěchá, má zájem si data pořádně prohlédnout. Měla by mu tedy být dána ta možnost.

Zde je možné si vzít inspiraci z přehrávání, které zná každý - tedy přehrávání záznamu z VHS/DVD. U těchto zařízení je k dispozici minimálně pauza. Ta se při přehrávání hodí i zde. Záznam půjde pozastavit a pořádně prohlédnout. Překlikávat mezi `Play` a `Pause` není příliš praktické. Bude tedy vhodné pauzu doplnit krokováním - na kliknutí načíst další řádek.

Další vlastnost hodna implementace je změna rychlosti přehrávání. V přijímači je možné dosáhnout nastavení, že na monitoru je ve vodopádu vidět sotva sekunda dat, je proto vhodné umět jejich přehrávání zpomalit. Stejně tak je vhodné umět rychlost přehrávání zvýšit, nejen pokud se aplikuje komprese dat při přehrávání - na čekání na přidání řádku sekundu, případně prohlížet dvoudenní záznam v reálném čase nemá trpělivost každý.

Stejně tak by byla zkouška trpělivosti nechat uživatele při hledání konkrétního času v půlce záznamu shlédnout záznam od začátku až po pozici, kterou hledal. Bude tak vhodné pro efektivní ovládání při dlouhých záznamech přidat ještě nějakou možnost skákat po větších časových úsecích, než jeden řádek, nebo rovnou přeskočit na požadovaný čas.

Pro úplnost analogie s DVD přehrávačem mezi ovládací prvky bude vhodné zařadit tlačítko Stop, pro zastavení přehrávání a vrácení se na začátek záznamu a Repeat, aby bylo snadné pouštět záznam stále dokola.



Obrázek 3.1. Přepracované přijímání a zpracování dat.

3.1.4 Timeshift

Nyní se dostávám k zajímavému oříšku. A co to vlastně timeshift je? Timeshift je funkce známá například z lepších DVB-T a satelitních přijímačů umožňující vracení se při sledování pořadu na dříve odvysílanou část. Hodí se v případě, když si jde člověk při reklamním bloku udělat čaj a zdrží se. Rovněž je tak možné přeskakovat reklamy, pokud se na pořad člověk dívat nějakou dobu po jeho začátku a reklamní bloky přeskočí přetočením do času blíže současnosti.

Ukládání dat je přímočaré. Mám napřijímaná data a umím je zobrazit na monitoru. Místo zobrazení na monitoru, je však uložím na pevný disk. Podobně přímočaré je to s přehráváním záznamu. Umím data zobrazit pokud jsou ve správném formátu a mám data na disku - jen je přečíst.

Timeshift je pak principielně kombinace obou těchto přístupů. Měření přijmu, zobrazím a pro prohlédnutí dat je nutné někam ukládat si dříve přijatá data, abych je mohl zobrazit.

V počítači jsou dvě použitelná úložiště - RAM a pevný disk. RAM exceluje v rychlosti a snadnosti přístupu. Data je možné uložit do nějaké formy kruhového bufferu a mít k nim tak rychlý a snadný přístup. Při zápisu na disk je nutné zvážit v jakém formátu data ukládat - podobně jako to bylo nutné při přehrávání.

Co se velikosti týče, běžný standard RAM je zhruba 8 GB, 16 GB už je v pracovních stanicích spíše nadstandard. To poslouží k uložení záznamu o délce zhruba hodiny při konzervativním nastavení přijímače. Pevný disk disponuje běžně kapacitou 1 TB, což vystačí na mnoho hodin.

Rychlost RAM je sice v desítkách GB/s, což je řádově více než pevný disk, ale rychlost přijímaných dat je ještě o řád menší než rychlost HDD (HDD cca 100 MB/s, data nejvýše 10 MB/s). Rychlost zde tedy není zásadní a data mohou být ukládána na pevný disk.

Rychlost disku se tedy zdá dostatečná a kapacitou jednoznačně vítězí pevný disk.

Pokud je tedy data nejen možné, ale i vhodné ukládat na pevný disk, jak to provést? Než znovuvynalézat kolo, stojí za zvážení, zda není možné využít schopnosti nahrávání a přehrávání dat. Data v záznamu k dispozici jsou. Obsahuji rovněž i informaci o tom, kdy byla pořízena. Výhodou je i možnost využít komprese.

Podívám se tedy na trasu, kudy putují data z UDP příjmu do zobrazovače. Někde v tomto řetězci 3.1 cestu rozpojím a vložím tam objekt starající se o přesměrování dat z příjmu do souboru a poté ze souboru do zobrazovačů.

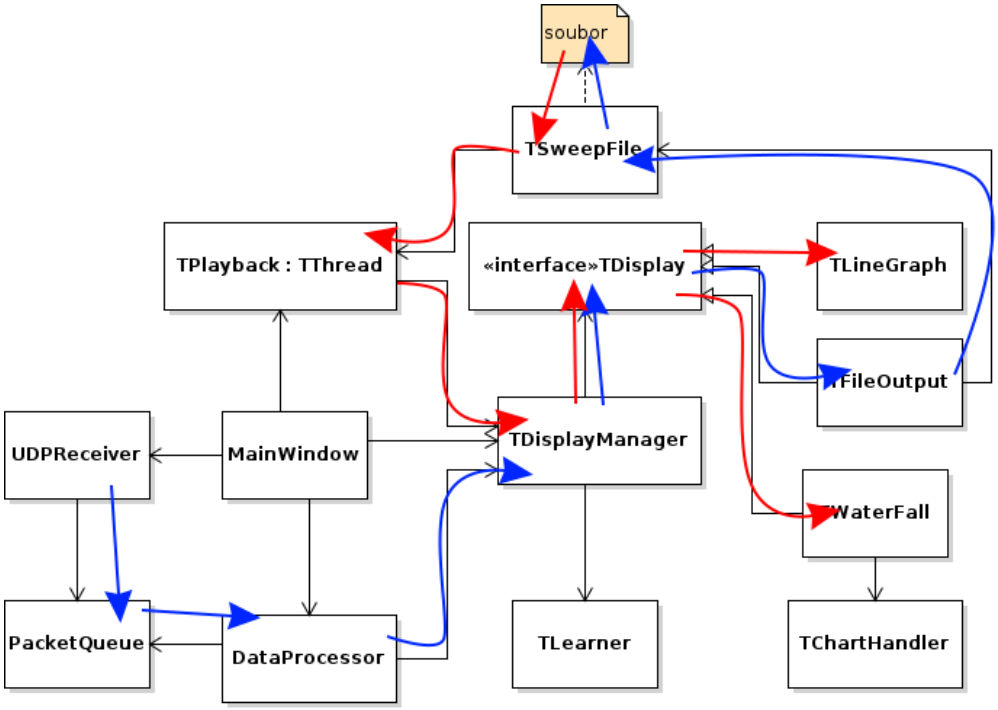
Cesta, kde se pomyslná cesta dat kříží je v `TDisplayManager`. Do něho přicházejí data z UDP a obsluhuje zobrazovače včetně toho, co ukládá data do souboru. `TDisplayManager` ale není uzpůsoben pracovat se dvěma různými toky dat. Raději však, než se tuto funkcionalitu snažit tam nějak doplnit, bude vhodnější vytvořit druhou instanci této třídy. Jedna bude sloužit k příjmu dat a ukládání do souboru. Druhá z toho samého souboru začne data načítat, jakmile tam budou uložena. Vznikne tak sice jisté zpoždění, ale to by nemělo být pro uživatele postřehnutelné.

Na obrázku 3.2 je znázorněn tok dat. Modrá ukazuje příjem dat z přijímače. Měření je zachyceno, zpracováno a uloženo stejně jako při běžném ukládání do souboru s tím rozdílem, že je pro fungování timeshift vyhrazena zvláštní(druhá) instance `TDisplayManager`. První instance funguje úplně stejně jako při přehrávání dat.

Takovéto uspořádání by mělo vytěžit ze stávajících tříd maximum bez nutnosti implementovat speciální třídy, která v podsatě jen nedokonale kopíruje možnosti tříd stávajících.

3.1.5 Implementace

Základem všech těchto tří úkonů - nahrávání, přehrávání a timeshift - je práce se souborem.



Obrázek 3.2. Přepracované přijímání a zpracování dat.

`TSweepFile` poskytuje jednoduché rozhraní pro práci se soubory. Tato třída umožňuje otevřít soubor v režimu čtení i zápisu.

Pro ukládání měření se musí zadat adresa cílového souboru. S prvním příchozím měření dojde nejdříve k vygenerování souboru s hlavičkou 3.1 a uloží se ono jedno měření. Následně se ukládají další a další podle toho, jak přicházejí.

Současně s ukládáním do souboru se zaznamenává kolik dat již bylo zapsáno. Tato informace je vhodná nejen pro uživatele, aby věděl, kolik dat již zaznamenal, ale nezbytná pro dělení souborů. K tomu dochází když velikost jednoho souboru překročí stanovenou hranici.

Nový soubor se vytvoří tak, že se k příponě prvního přidá pořadové číslo souboru. Po delším měření pak cílová složka může vypadat nějak takto:

```
měření.cap
měření.cap1
měření.cap2
...
měření.cap11
...
```

Když dojde k vytvoření dalšího souboru s pořadovým číslem, uloží se do něho hlavička stejně jako do prvního souboru, zavře se soubor, jenž dosáhl maximální velikosti a pokračuje se s ukládáním jednotlivých měření. Když je měření dokončeno, zavře se i poslední soubor.

Při přehrávání dochází k otevření prvního souboru s názvem `*.cap`. Název slouží jako reference základního názvu pro generování ostatních. Při otevření prvního souboru se zkontroluje validnost hlavičky - ta musí obsahovat validní magickou konstantu a verzi.

`TSweepFile` obsahuje funkci pro získání jak samotné hlavičky, aby se podle ní mohlo nakonfigurovat uživatelské rozhraní, ale i kombinaci hlavičky v kombinaci s měřením, kvůli přehrávání.

Protože je měření rozprostřeno přes více souboru, jsou informace o jednotlivých souborech uloženy v poli struktur `TSweepFileBit`. To umožňuje rychlejší přístup k jednou již otevřeným souborům.

```
struct TSweepFileBit{
    FILE * File;
    String Name;
    CaptureFileHeader Header;
    long long Size;
};
```

Původně jsem chtěl kvůli rychlosti nechávat jednotlivé soubory neustále otevřené. MSDN dokumentace[7] tvdí, že pomocí funkce `fopen()` je možné mít otevřeno až 512 souborů (resp. 2048 po zvětšení limitu funkcí `_setmaxstdio()`), což je počet více než dostatečný. Tomuto číslu se mi však nepovedlo ani přiblížit(limit byl cca 30) a tak při otevření nového dojde k zavření předchozího.

Další z dostupných funkcí je získání času posledního uloženého měření. V té dochází k pokusu o otevření co nejposlednějšího souboru a následně projití až k poslednímu měření v něm uloženém. Vedlejším efektem této funkce je aktualizace seznamu souborů daného měření v případě, že přibyl další. Této vlastnosti je využíváno hlavně u funkce timeshift.

Nezbytnou součástí je kvůli přehrávání také možnost přesunout se na libovolnou pozici v měření. To je realizováno funkcí `bool SeekTime(TDateTime SeekTime)`. Parametrem je čas, kdy bylo požadované měření realizováno. Při jejím zavolání dojde k prohledání souborů a přesunu na nejbližší předchozí měření - to je pak k dispozici při následujícím zavoláním `GetSweep()` vracející měření spolu s jeho hlavičkou. Při dalším volání se v načítání postupuje od tohoto měření dále a původní pozice je tak zapomenuta.

3.1.6 TFileOutput

Rozhraní pro práci se souborem je již dokončeno, mohu ho začít využívat k ukládání měření. K tomu slouží `TFileOutput`. Tato třída je potomkem TDisplay, takže se řadí mezi vodopád a spektrum a stejně jako oni je i on rovněž spravovaný v `TDisplayManager`.

Po jeho vytvoření je možné spustit, nebo zastavit měření zavoláním funkcí `StartRecording()` a `StopRecording()`. Po spuštění se vyvolá dialog vyzývající uživatele k zadání cesty kam se má nahrávka uložit. Pokud se zastaví nahrávání a nedojde ke zrušení objektu, nevyvolá se při opětovném spuštění dialog pro vybrání adresy a dojde k přepsání souborů. Instance této třídy je tedy na jedno použití.

`Save dialog` je dostupný pomocí API Windows[8]. Rychle tak získám přívětivý a uživatelům známý způsob vybírání souboru.

Kromě samotného získání názvu souboru poskytuje mnoho možností jak si ho personalizovat podle vlastních potřeb a ulehčit tak život uživateli i sobě.

Pro konfiguraci tohoto dialogu slouží struktura `OPENFILENAME`[9]. Do té se buď jako číslo, nebo jako textový řetězec uloží parametry jednotlivých konfigurací. Nejdříve strukturu inicializuji:

```
OPENFILENAME ofn ;
ZeroMemory(&ofn, sizeof(ofn));
ofn.lStructSize = sizeof(ofn);
ofn.hwndOwner = NULL;
```

První věcí, kterou využiji je filtrování podle přípon. Protože ukládám záznam, zvolím příponu `.cap` - zkráceně `capture`. Zvolit nějakou konkrétní příponu a nenechat to pouze na uživateli je vhodné, ne-li dokonce nezbytné, protože od nástupu Windows 7, je přípona ve výchozím nastavení skryta a název souboru tak na první pohled nemusí poskytovat informaci o tom, že je tento soubor skutečně vhodný k přehrávání v mém programu.

```
ofn.lpstrFilter = (LPCWSTR)L"IP982 CaptureFile (*.cap)\0*.cap\0All
Files,(*.*)\0*.*\0";
```

Do `ofn.lpstrFilter` ukládáme informaci o příponách, které mají být viditelné ve formátu:

```
"Název Filtru", znak NULL,"parametry filtru", znak NULL
```

Název filtru je to, co vidí uživatel v roletce nad tlačítkem Save. Parametry filtru jsou informace pro dialog. Tvořen může být jakýmkoliv platným znakem názvu souboru, nebo hvězdičkou, značící jakékoliv množství a opakování platných znaků. Filtrem `*.cap` tak zobrazím pouze soubory s příponou `.cap`.

```
ofn.lpstrFile = (LPWSTR)szFileName;
ofn.nMaxFile = MAX_PATH;
```

Do `lsptrFile` se ukládá pointer na mnou připraveného pole znaků - Cčkového řetězce - sloužícího k uložení cesty k vybranému souboru. A proto, že první znak tohoto bufferu musí být nul, je nutné do `nMaxFile` předat i informaci o tom, jak dlouhý buffer je k dispozici. V případě příliš krátkého bufferu nedojde k vyhození výjimky, ani pádu programu, ale funkce otevírající okno vrátí `false` a parametr `CommDlgExtendedError` struktury se uloží konstanta `FNERR_BUFFERTOOSMALL`. Je tedy nutné si připravit buffer dostatečně velký, aby uživatel nebyl zatěžován opakovaným vybíráním souboru.

```
ofn.Flags = OFN_PATHMUSTEXIST | OFN_HIDEREADONLY;
```

Díky `Flags` se získá mnoho drobných nastavení usnadňující práci nejen uživateli ale hlavně i vývojáři. Zde jsou však použity jen `OFN_PATHMUSTEXIST`, aby uživatel nemohl zadat neexistující cestu a `OFN_HIDEREADONLY` schovávající soubory pouze pro čtení, do kterých nejde zapisovat.

```
ofn.lpstrDefExt = (LPCWSTR)L"cap";
```

Parametr `lpstrDefExt` pak už slouží pouze pro pohodlnost vývojáře. Pokud totiž uživatel při zadávání názvu souboru nepřidá správnou koncovku, přidá se automaticky - není tak nutné kontrolovat správnost tvaru

Když jsou parametry struktury `OPENFILENAME` nastavené, je možné zavolat `GetSaveFileName(&ofn)`, která vyvolá onen dialog pro zadání cesty k cílovému souboru.

Pokud uživatel vybere soubor korektně, pak je návratová hodnota funkce `GetSaveFileName()` rovna `false` a vice versa. Při neúspěchu lze zavolat funkci `CommDlgExtendedError()`, která vrací kód chyby, která nastala. V tomto smyslu může být chybou i stisknutí tlačítka Storno, nebo zavření dialogu.

3.1.7 TPlayback

Tato třída se stará o přehrávání souborů. Její náplň je ekvivalentní náplni tříd `UDPReceiver` a `PacketProcessor` a ze stejného důvodu také běží ve vlastním vlákně. Tím důvodem je neblokování grafického rozhraní. Načítání z pevného disku a zpracování měření je časově náročná operace a spouštění načtení a zpracování z hlavního vlákna aplikace by tak způsobovalo velmi nepříjemné latence programu.

Vlákno běží v téměř nekonečné smyčce(při ukončení přehrávání se smyčka ukončí). V každém jejím cyklu dochází ke čtení řídících proměnných (o nich dále) a vykonávání příslušných akcí a načtení jednoho měření ze souboru a předání zobrazovačům přes `TDisplayManager`.

Pro ukončení smyčky vlákna slouží proměnná `FTerminate`. Ta je při inicializaci nastavena na `false` a protože smyčka má podobu `while(!FTerminate){...}`, je jejím nastavením na `true` ukončena. Následuje uzavření souboru a uvolnění paměti.

K souboru je přistupováno pomocí rozhraní poskytovaného třídou `TSweepFile`. Není tedy nutné vědět jak soubor vypadá tím se celá třída podstatně zjednodušuje. Celý přístup do souboru je omezen na několik jednoduchých akcí. Otevři a zavři soubor, načti konfiguraci měření, načti měření s hlavičkou, zjisti od kdy do kdy byl soubour nahráván, posuň se na určenou pozici.

Protože dochází k ovládání jednoho vlákna z jiného, bylo nutné vybrat některý ze synchronizačních mechanismů. Zde se jedná o v podstatě jednosměrné ovládání a proto postačí proměnné. Hlavní z nich představuje příkaz, co má být uděláno. Je reprezentována výčtovým datovým typem a může nabývat těchto hodnot:

```
enum RecordingActions {raNone, raPlay, raStop, raPause, raPauseStep,
raSelectFile, raOpenWithFilename, raLoopEnable, raLoopDisable};
```

A zde je jejich význam:

- `raNone` je výchozí hodnota. Pokud je nastavena, je možné přiřadit další řídící příkaz.
- `raPlay` spustí přehrávání/ruší pauzu. Není-li žádné otevřené, vyvolá se open dialog pro vybrání souboru.
- `raStop` zastaví přehrávání a vrátí se na začátek souboru/ů.
- `raPause` pozastaví přehrávání - drží aktuální pozici.
- `raPauseStep` pokud je aktivní pauza přidá další jedno měření.
- `raSelectFile` vyvolá open dialog pro vybrání souboru k otevření
- `raOpenWithFilename` začne přehrávat bez vyvolání dialogu. Podmínkou je, že nastavení této akce musí předcházet nastavení názvu souboru k přehrávání.
- `raLoopEnable` aktivuje přehrávání ve smyčce. Pokud tedy přehrávání narazí na konec souboru, začne se přehrávat od začátku.
- `raLoopDisable` deaktivuje přehrávání ve smyčce.

Mechanismus mezivláknového řízení pomocí proměnné funguje tak, že třída, jež má být řízena obsahuje jednu proměnnou. Tu periodicky čte a pokud se změní, vlákno se zachová dle příkazu.

Aby bylo zajištěno, že komunikace skutečně probíhá, je zde `raNone`. Ta je nastavena v výchozím stavu a znamená žádnou akci. Řízené vlákno tak ví, že se nejedná o příkaz a řídící, že může nějaký příkaz přiřadit. Pokud to udělá, řízené vlákno při svém dalším čtení zjistí, že se hodnota změnila a příkaz vykoná, následně opět

přiřadí hodnotu `raNone`. Řídící vlákno zároveň před přiřazením nového příkazu čeká, dokud není hodnota proměnné znovu v jejím výchozím stavu.

Další proměnou je `PlaybackSpeed`. Jak již název napovídá, slouží k řízení rychlosti přehrávání. Při normální rychlosti je měření načítáno s takovými intervaly, s kterými bylo pořízeno přijímačem. To může být v závislosti na nastavení jednotky až stovky milisekund. Pokud je využito agregace, může být interval klidně i desítky sekund.

Pomocí `PlaybackSpeed` tak lze nastavit kolikrát se má tento interval prodloužit, případně zkrátit, aby bylo možné sledovat záznam takovou rychlostí, při které to má smysl.

Hodnota je uložena v datovém typu double. Ten může dosahovat i hodnoty `Infinity`, tedy nekonečno. Zde je tato hodnota použita pro stav, kdy se mezi přidáním jednotlivých měření nečeká a načítání a zpracování probíhá tak rychle, jak to hardware dovolí.

Když je potřeba posunout se v souboru na námi požadovanou pozici, musí se třída dozvědět kam přesně se přesunout má. K tomu se používá funkce `SetSeekTime(TDateTime time)`. Ta nastaví do jakého času se má přehrávání přesunout. Samotné přesunutí v souboru je vykonáno ve smyčce `TPlayback` za situace, kdy je v privátní proměnné `FSeekTime` upravené ve funkci `SetSeekTime()` nenulová hodnota.

Postup započetí přehrávání je poměrně jednoduchý. Po vytvoření třídy `TPlayback` je nejprve nutné nastavit správce zobrazovačů (`TDisplayManager`). Následuje otevření konkrétního souboru spuštění přehrávání a to může probíhat třemi způsoby:

1. Název souboru je znám - nastavíme jej pomocí `SetupFilename()`, zvolíme akci `raOpenWithFilename`
2. Název souboru znám není...

   I ...a chceme začít přehrávat později - zvolíme `raSelectFile`. Dojde k vyvolání open dialogu, kde uživatel vybere, který soubor chce otevřít. Kdykoliv je pak možné nastavit `raPlay` a spustit tak načítání měření.

   II ...a chceme začít přehrávat okamžitě - zvolíme `raPlay`. Tím, že rovnou zvolíme tuto volbu dojde ke spuštění přehrávání. Není však čeho a je tak vyvolán open dialog pro otevření souboru. Okamžitě po otevření se začnou načítat data.

Podobně jako u `TFileOutput`, kde je pro otevírání souborů použitý dialog poskytovaný Windows API (save dialog), je i zde otevírání řešeno přes Windows API. Postup inicalizace je shodný až na použité příznaky, je vypuštěn `OFN_HIDEREADONLY`, jelikož soubor otevíráme v pouze pro čtení. Pokud je struktura `OPENFILENAME` inicializována, je dialog vyvolán skrze zavolání funkce `GetSaveFileName(&ofn)` namísto `GetOpenFileName(&ofn)`.

Přehrávání souboru pak probíhá postupným načítáním jednotlivých měření a jejich dávkováním do zobrazovačů. Čas mezi předáním jednotlivých měření je určen časem jejich pořízení a řídící proměnnou `PlaybackSpeed`. Tou se násobí a tím zvětšuje, nebo zmenšuje jejich prodleva odpovídající rozdílů časů jejich nahrání.

Aby měl uživatel nějakou zpětnou vazbu o stavu přehrávání, je i zde použita fronta událostí `EventQueue`. `TPlayback` může vyvolat 3 události:

- `PLAYBACKTIME` - slouží k předání časové informace. Ta je reprezentována doublem. V int-ové části, je pak zakódováno o který čas se jedná
  - 0: čas pořízení naposledy načteného měření
  - 1: čas pořízení prvního měření
  - 2: čas pořízení posledního měření
- `PLAYBACKFILEOPENFAILED` - nepovedlo se vybrat/otevřít soubor

Událost `PLAYBACKTIME` je při otevření souboru postupně vyvolána se všemi třemi parametry. Hlavní okno tak může uživateli zobrazit detaily souboru.

3.1.8 Uživatelské rozhraní

Žádný program by nebyl kompletní bez uživatelského rozhraní. Pro ovládání přehrávání a nahrávání jsem vycházel z principu návrhu grafického rozhraní `Recognition over recall`, který konstatuje, že uživatel se rychleji orientuje, pokud rozpozná dříve viděné, namísto toho, aby si snažil vzpomenout jak se konkrétního cíle dosahuje. Vytvořil jsem proto sadu ikon na obrázku 3.3, které jsou běžně k vidění na přehrávačích videa, hudby a jiných multimédií a přiřadil jsem jim ekvivalentní funkcionalitu. Uživatel tak bude hned vědět, že rovnoramenný trojúhelník směřující doprava slouží ke spuštění přehrávání, dvě svislé rovnoběžky k pozastavení, atd.

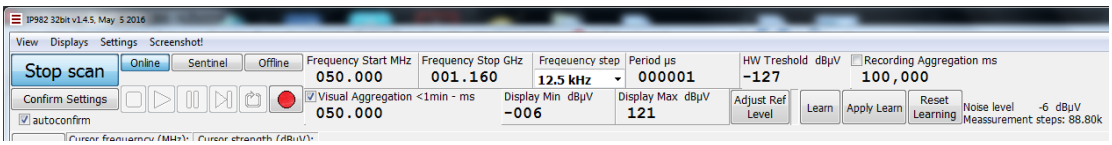
Obrázek 3.3. Ikony ovládání přehrávání a nahrávání.

Tyto prvky jsou umístěné na panelu mezi ostatními a jsou de/aktivovány, podle toho v jakém módu je program provozován. Při deaktivaci ovládacího prvku dochází k jeho zešedivění a při stisknutí, nebo jiné akci nejsou volány callbacky a tedy jako by ani nebylo - byť má uživatel šanci ho vidět. Rozdíl mezi aktivovanými a deaktivovanými tlačítky je vidět na obrázcích 3.5 a 3.8.

Zmíněnými módy jsou jsou tři:

- Online
- Sentinel
- Offline

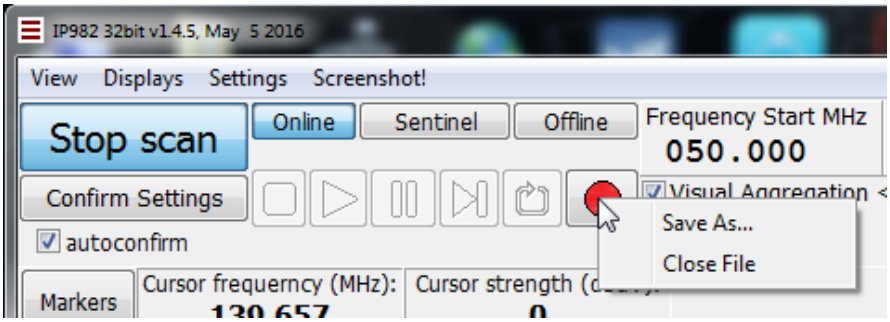
A je jimi myšleno to, jak bude bude program využíván z hlediska práce s daty. Jejich specifikaci představím v následujících odstavcích.



Obrázek 3.4. Ovládací prvky při aktivním Online módu.

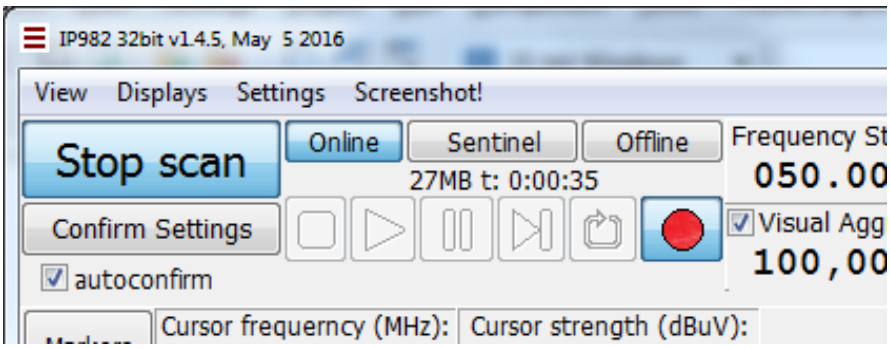
`Online` mód slouží pro je přímé zobrazení. Je možné nakonfigurovat parametry měření přijímače, či spustit nahrávání. To je možné spustit buď kliknutím na tlačítko nahrávání, nebo využít kontextového menu (Obrázek 3.5). S jeho pomocí je možné zvolit si cílový soubor před započetím nahrávání (volba `Save As`). Bez tohoto kroku je po stisknutí tlačítka před započetím zobrazen dialog pro výběr cíle ukládání.

Ovládač s popiskem `Recording Aggregation` slouží k nastavení agregace při nahrávání - tedy jak často se mají zaznamenávat řádky měření, aby se snížila datová náročnost. Volba je možné rychle deaktivovat zaškrtávátkem v levém horním rohu prvku.



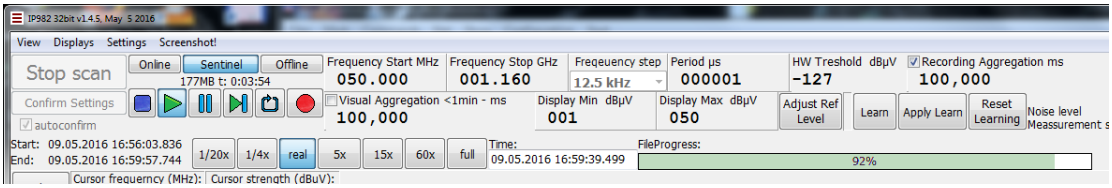
Obrázek 3.5. Přepracované přijímání a zpracování dat.

Průběh nahrávání je zobrazován mezi tlačítky pro přepínání módů a ovládání přehrávání. Je zobrazena celková velikost souborů a délka trvání nahrávky.



Obrázek 3.6. Stav nahrávání.

Pro ukončení nahrávání je možné buď opět využít kontextového menu a zvolit `Close File`, nebo stisknout nahrávací tlačítko znovu. Pokud nedojde k uzavření souboru, nebo vybrání nového cíle, dojde při opětovném stisku tlačítka nahrávání k přepsání původního souboru.

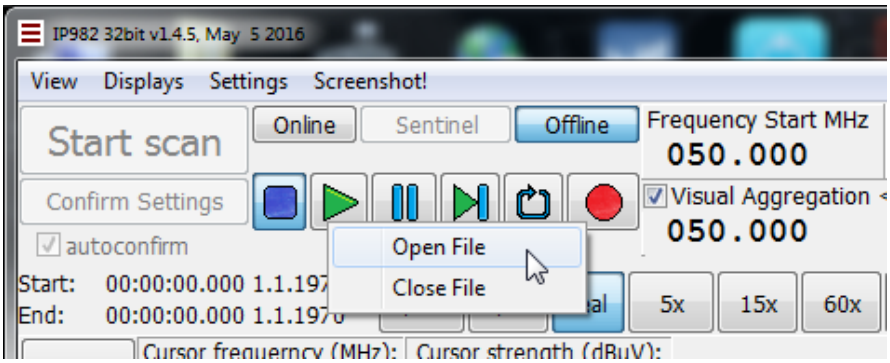


Obrázek 3.7. Mód `Sentinel` kombinuje `Online` i `Offline` mód.

Do módu `Sentinel` je možné přepnout pouze v případě, že je aktivní `Online` mód a spuštěno měření. To má svůj důvod. Při nahrávání není možné změnit nastavení a tak je při přepnutí z `Online` do `Sentinel` nastavení zmrazeno a jsou deaktivovány ovládací prvky měnící toto nastavení. V momentě přepnutí do totho módu dojde také k vyvolání dialogového okna pro výběr cílového souboru. Tento soubor nejenže bude využit pro timeshift, ale kromě toho zůstane stejně, jako jakákoliv jiná nahrávka, uložena na disku s možností pozdějšího přehrání.

Timeshift je realizován přehráváním právě zaznamenávaného souboru. Tím dochází k neustálému narůstání koncového času záznamu, ten je však pravidelně aktualizován, aby byla reflektována skutečnost. Tato aktualizace se projeví jak na popisku, tak na progressbaru. Ovládání timeshift je stejné jako ovládání přehrávání při `Offline` módu (od kterého se mód `Sentinel` liší pouze tím, že nejsou současně do souboru zapisována nová data) a tím se tak k němu dostávám.

V `Offline` módu jsou přehrávány nahrávky, které byly pořízeny dříve a to buď v módu `Online`, `Sentinel`, nebo dokonce `Offline`. Přehrávání je možné aktivovat dvěma způsoby podobně tomu, jak je to zařízeno při nahrávání. První je pouhé stisknutí tlačítka pro přehrávání, následně dojde k vyvolání open dialogu pro výběr souboru po jehož potvrzení dojde okamžitě k započetí přehrává. Druhý je vyvolání kontextové nabídky (Obrázek 3.8) a zvolení `Open File` dojde k vyvolání openfile dialogu. Spuštění přehrávání vybraného souboru začne stisknutím tlačítka přehrávání.



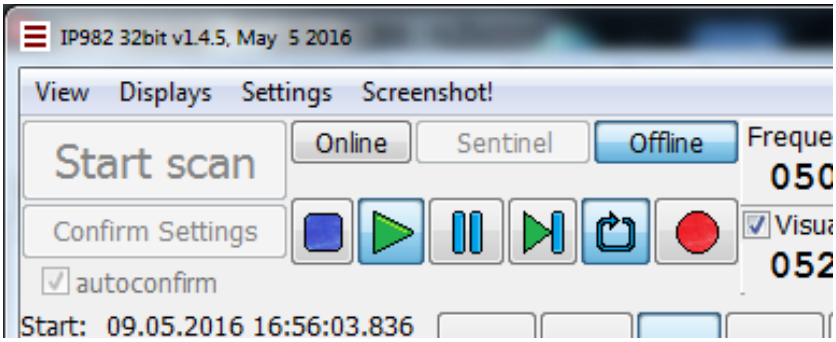
Obrázek 3.8. Otevírání souboru pro přehrávání.

Přehrávání je možné zastavit tlačítkem Stop. Kromě zastavení tímto dojde i k navrácení na začátek souboru k prvnímu měření.

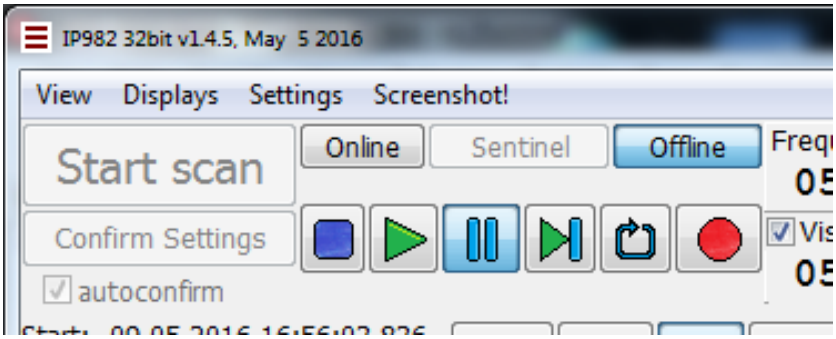
Na obrázku 3.9 je kromě přehrávání aktivní i přehrávání ve smyčce. Jakmile se tedy do zobrazovačů předá poslední měření, dojde k opětovnému přehrávání od začátku, jako by záznam pokračoval do nekonečna.

K pozastavení přehrávání slouží tlačítko Pauza. Jeho stiskem se pouze přestanou přidávat nová měření, pozice přehrávání zůstane zachována. Pokračování přehrávání započne po stisknutí tlačítka Přehrávání. Při pauze je možné přidávat měření po jednotlivých měření klikáním na tlačítko Krok.

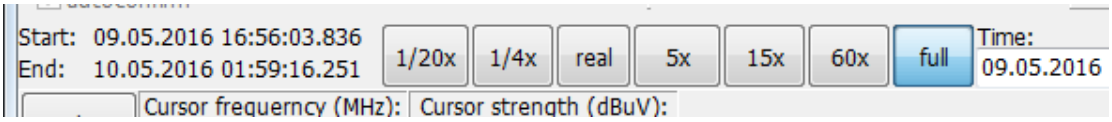
Ovládací panel začnu popisovat zleva (Obrázek 3.11, celý je pak k vidění na obrázku 3.7). Nejdříve jsou informace o tom, kdy byl záznam pořízen. Start a End uvádí čas, kdy bylo zaznamenáno první a poslední měření. Tlačítky s popisky 1/20x, 1/4x, real, 5x, 15x, 60x a full se řídí rychlost přehrávání. Rychlost real, odpovídá tomu, jak byl záznam původně pořízen. Při rychlosti full, jsou ignorovány



Obrázek 3.9. Přehrávání ve smyčce.



Obrázek 3.10. Pauza v přehrávání.



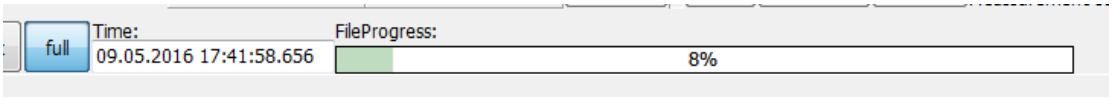
Obrázek 3.11. Ovládací panel přehrávání(levá část).

časové rozdíly mezi jednotlivými měřeními, podle kterých je normálně přehrávání řízeno, a měření jsou uživateli servírována takovou rychlostí, kterou je použitý počítač dosáhnout v závislosti na jeho výkonu.

Na pravé části úplně v pravo je ukazatel průběhu File Progress, který znázorňuje v jaké pozici vzhledem k počátku a konci měření je naposledy přidané měření. Prvek s názvem Time je čas pořízení naposledy přidaného měření. Tento prvek má ještě jeden účel. S jeho pomocí se uživatel v záznamu může přesunout v záznamu kam chce.

Hodnota tohoto prvku je měněna s každým přidaným měřením a tak, jakmile se do něho klikne, toto obnovování skončí a je zastaveno na hodnotě, která zde byla při poslední obnově a tedy momentu, kdy do něho uživatel klikl. Přehrávání zastaveno není. Následně může hodnotu přepsat na nějakou, mezi časy Start a End. Následným stisknutím klávesy `Enter`, dojde k potvrzení a program se pokusí soubor 'přetočit' na požadovanou pozici. Při přetočení jsou je vodopád vyresetován, aby uživatele nemátly nesouvislé měření. Pokud tento čas není v uvedeném intervalu, je v přehrávání pokračováno beze změny. Může také kliknout mimo tento prvek. Tím dojde k zrušení případných změn a opětovnému obnovování aktuálního času.

Dříve jsem zmínil, že možnost nahrávání v `Offline` módu. Nahrávání se ovládá stejně jako při `Online` módu. Důvodem proč použít v režimu, kdy dochází k pře-



Obrázek 3.12. Ovládací panel přehrávání(pravá část).

hrávání záznamu, zaznamenávání může být vícero. Od zvýšení komprese přehráním celého záznamu a uložení s větší agregací, až po získání záznamu obsahující jen několik specifických měření.

Kapitola 4
Detekce

Při detekci vycházím z premisy, že signál je zajímavý pouze pokud se objeví, nikoliv pokud zmizí. Můžeme mít zařízení které je rušeno, víme kdy je jeho činnost narušena, avšak nejsme si jistí čím. Můžeme vědět, že zařízení je rušeno a dokonce i na jaké frekvenci, ale nemusíme vědět kdy.

Můžeme mít rovněž zařízení, které má výpadky ve vysílání a přitom přímo z přístroje nemusí být zjistitelné, kdy k těmto výpadkům dochází.

Dalším způsob užití je detekce vysílání v licencovaném pásmu. V místě, kde máme podezření na tuto činnost provedeme učení a následně necháme přístroj detekovat, zda se toto děje.

Nějakou formou detekce můžeme rovněž testovat, zda námi sestrojené zařízení v nějakých svých fázích činnosti nevysílá rušivé frekvence a otestovat tak oprávněnost Prohlášení o shodě, které musejí mít všechny (nejen)elektronické výrobky prodávané u nás i v Evropě.

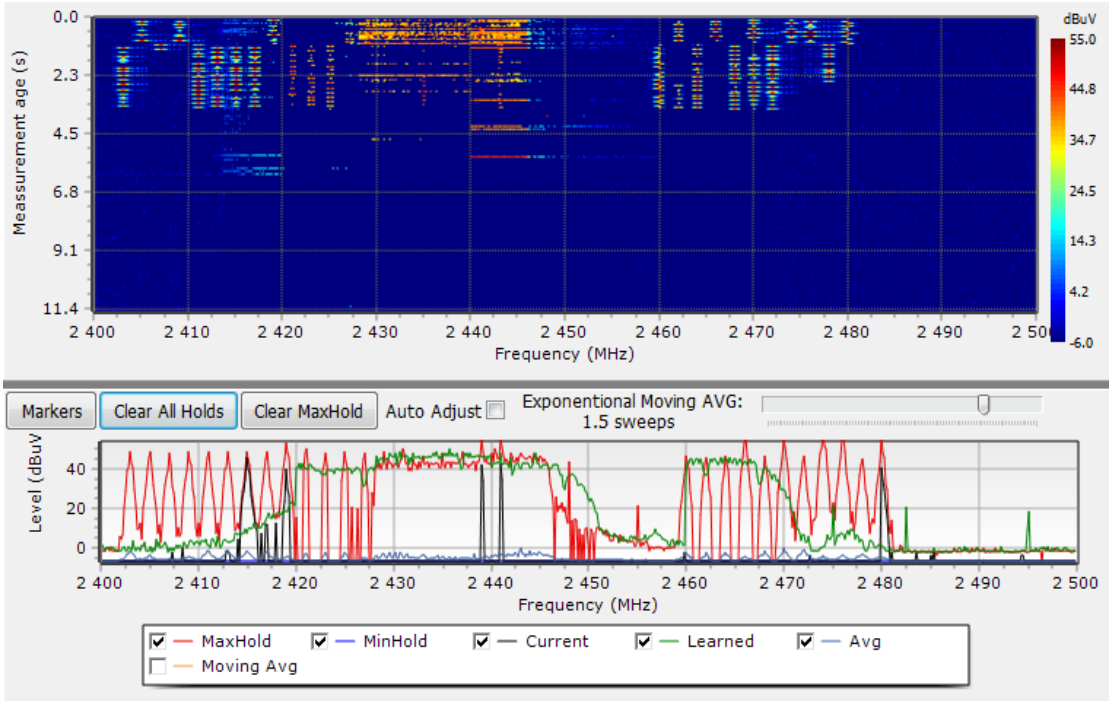
Z těchto případů užití vychází, že zajímavá je především situace, kdy nějaký signál přibude oproti klidové situaci (vypnutý přístroj, korektně fungující přístroj). Použití pro opačnou situaci se však také evidentně najde.

V rámci bakalářské práce jsem v rámci zobrazení spektra implementoval funkcionalitu naučení maximálních hodnot a jejich odstranění ze zobrazení ve spektru a vodopádu. Už toto by se dalo nazvat detekcí. Na obrázku 4.1 jsem nejdříve zapnul učení pro frekvenční rozsah 2,4 GHz až 2,5 GHz. Tento rozsah je bezlicenční a tak je hojně využíván. Autíčky na dálkové ovládání počínaje, poskytováním internetového připojení konče. Učení je pak jednoduché uchovávání maximální dosažené úrovně pro danou frekvenci. Na zobrazení spektra je toto učení znázorněno zeleně. Po ukončení učení jsem resetoval MaxHold a chvíli na to zapnul bezdrátový Bluetooth reproduktor. Technologie Bluetooth využívá FFHS (Frequency Hopping Spread Spectrum), pro snížení rušení.

V mém okolí byly při měření aktivní WiFi zařízení komunikující na několika kanálech. To se při učení projevilo zvýšením naučené úrovně nad šum na frekvencích 2,42 GHz až 2,45 GHz a 2,46 GHz až 2,47 GHz. Naučením těchto hodnot došlo k jejich efektivnímu odstranění ze spektra. Protože jsem učení prováděl poměrně krátce, došlo k tomu, že se nějaké signály dostaly přes naučenou úroveň a tím i do spektra. (K tomu došlo v čase mezi 4,5 a 6,8 sekundy.). Mám tedy principelně naučenou klidovou hodnotu.

Po zapnutí reproduktoru dochází okamžitě k prohledávání okolí, zda se v něm nenachází jiné zařízení, se kterým by se mohl spárovat. Toto rozhodně neodpovídá klidovému stavu a tak se to projeví na vodopádu a spektru jako rovnoměrně rozložené špičky.

Detekce by tedy mohla být založena na tomto principu. Na začátku se naučím klidový stav. Bude jasné, kde se vysílá a kde ne. Následně učení vypnu a začnu



Obrázek 4.1. Zkouška učení.

zaznamenávat signály které překročí naučené hodnoty. Tyto malé události se pak zkusím pospojovat a analyzovat je.

Výhodou tohoto přímočarého přístupu je, že takto mohu zjistit nejen signály, které se objevily, ale i ty, které zmizely. Pokud budu při měření záznamy ukládat, mohu se naučit na novějším, pustit detekci na starém a získat tak ty signály, které dříve byly a nyní nejsou.

Detekce je rozdělena na dvě části. První se stará o detekci jednotných událostí. Událost je definována tak, že jde o překročení naučené úrovně po nějakou dobu. Ke každé takovéto události budu ukládat frekvenční parametry a nejvyšší úroveň.

Tyto události následně vezmu a spojím do jednoho souboru podle společných vlastností a zkusit z těchto souborů získat nějaká zajímavá čísla.

Rychlý pokus s reproduktorem ukázal, že i přes učení, které by mělo odfiltrovat standardní pozadí může občas vykouknout zpět. Při detekci by toto vykouknutí způsobilo falešně pozitivní detekci. Jedním způsobem jak s tím bojovat je mít dostatečně dlouhou dobu učení. Tím, že bude probíhat dlouho by mělo dojít ke všem možným situacím a zachytit ty opravdu největší úrovně. Tato metoda má však vady. Trvá dlouho a není spolehlivá. Úroveň může vzrůst například odrazem od nějaké dočasné překážky. A to dlouho po tom, co ukončím učení.

Je nepravděpodobné, že by po dostatečně dlouhém učení úroveň takovýchto překročení dosahovala nějak velkého rozdílu. Profil úrovní by měl víceméně odpovídat, mohou se však o něco zvýšit. Pokud tedy vím, nebo předpokládám, že učení není dostatečně dlouhé, mohu zvýšit celou naučenou hodnotu o pár jednotek a eliminovat mírné kolísání maximální úrovně i při krátkém učení.

Stejně tak jako minimální rozdíl i případné překročení v pouze jedné z naměřených hodnot, může dávat falešně pozitivní detekce. V dnešní době se hojně využívá modulace, která šířku signálu rozšíří na nějaké pásmu. FM rádio má šířku modulace desítky kHZ. Mobilní operátoři využívají kolem 8 MHz. Kromě minimálního

překročení je tak vhodné dát možnost nastavit minimální šířku detekovaného signálu.

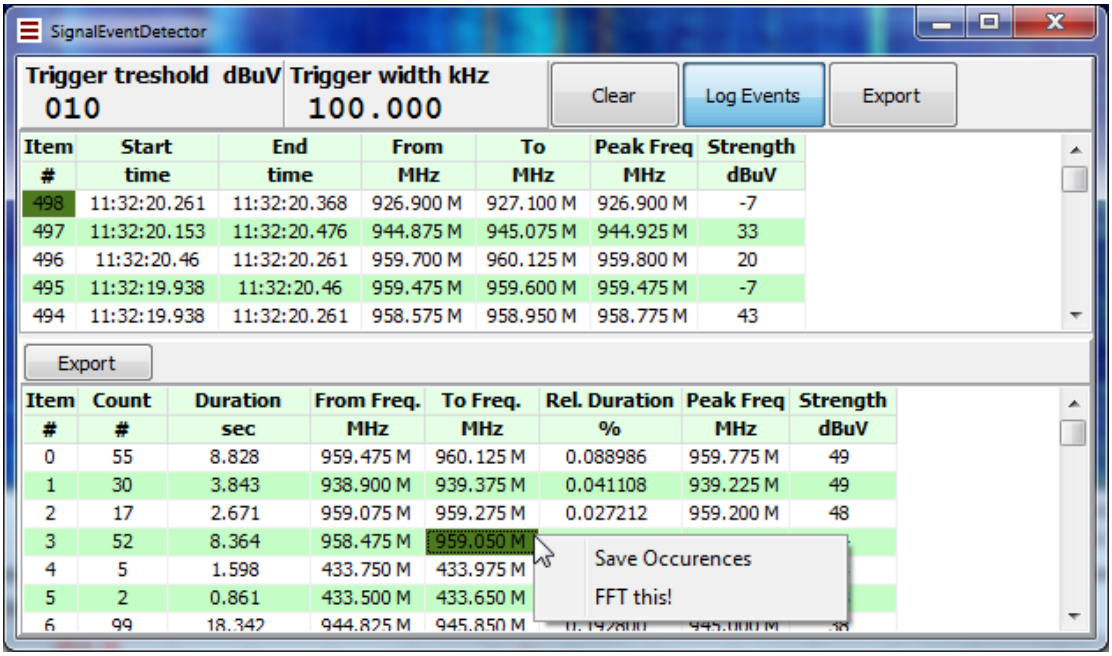
Detekce rozdělím na dvě logicky oddělené části. První část bude mít na starost detekcí událostí. Událost je překročení naučené úrovně o zadané parametry. Druhá vezme tyto události a sloučí je dohromady. Na těchto balíčcích pak bude prováděna analýza.

Informace, které se z těchto událostí dají vyčíst jsou časy toho, jak dlouho události trvaly a na jakých se vyskytly frekvencích. Ještě zajímavější je informace o tom, jak se tyto události opakují v čase. Kromě možnosti vykreslit průběh toho, jak se vyskytují a s jakou úrovní signálu. Pro analýzu tohoto průběhu se nejlépe hodí DFFT. Pro tuto analýzu použiji knihovnu FFTW[10].

4.1 Implementace

Detektor pracuje s naměřenými spektry a tak je i on založen na třídě `TDisplay`. Kromě měření musí být pro tento zobrazovač nakonfigurována i křivka učení, aby bylo s čím porovnávat.

Pro přehlednost jsem umístil vizualizaci obou analyzačních částí na jeden panel, resp. do jednoho okna (viz obrázek 4.2). Obě části mají shodně výstup ve formě tabulkového výpisu a nad ním menu pro ovládací prvky. Mezi nimi je dělič, který slouží ke změně poměru mezi oběma částmi tak jak je možné k vidění v mnoha IDE, nebo editoru fotografií, kde je možné změnit velikost nástrojových panelů.



Obrázek 4.2. Detektor.

Horní část je pro detekci událostí. `Trigger treshold dBuV` nastavuje o kolik musí být úroveň vyšší než naučená. `Trigger width kHz` udává minimální šířku překročení. Pomocí tlačítka `Clear` dojde k celkovému vyčištění všech událostí a jejich balíků - agregovaných událostí. `Log Events` aktivuje detekci za předopkladu,

že je systém naučený. `Export` v obou případech slouží k vyexportování všech řádků příslušné tabulky.

U agregovaných událostí je možné vyexportovat konkrétní agregovanou událost tak, aby bylo možné data načíst a zpracovat v jiném programu - například Matlabu. Tento export se provádí vybráním konkrétního řádku a vyvoláním kontextové nabídky. Z této nabídky jde rovněž provést DFFT.

Pro export dat jsem zvolil formu CSV. Je vhodnější než XML, protože data jsou ve formě tabulky, která má každý řádek stejně formátovaný. XML, které disponuje tagy a stromovou strukturou je zbytečně komplexní. Tagy v zněkolikanásobí velikost (pro zaznamenání síly signálu by musel vypadat zápis takto `<strength>12</strength>` namísto tohoto `12,` - tedy skoro desetinásobek znaků). Výhodou CSV je i snazší parsování. Microsoft Excel na to má jednoduchého průvodce, v Pythonu se na každý řádek zavolá `split(';')`, Matlab má `csvread()`,...

Data jsou vkládána do CSV ve stejném formátu, jako

```
item#; begin time; end time; total duration; relative duration; ...
0;25;15:59:26;15:59:54;1.348000;0.047239;1890400000.000000;1893600000...
1;46;15:59:26;15:59:55;4.433000;0.153031;845600000.000000;851000000...
2;10;15:59:26;15:59:54;0.540000;0.019250;938200000.000000;939600000...
...
```

Detekce je prováděna při získání měření. Na každé frekvenci měření porovnávám s naučenými hodnotami. Nejdříve jsou porovnávány úrovně. Pokud je vyšší než ta naučená o zadaný rozdíl, najdu pozici, kde úroveň začne být překračována a kde přestane. Z tohoto rozdílu a frekvenčního kroku spočítám šířku překročení v hertzích. Pokud je širší než zadaná minimální šířka, vznikne událost.

Tato událost je porovnána s probíhajícími událostmi a pokud je nalezena shoda, jsou tyto sloučeny. Upraví se rozsah, aby reflektoval nejnižší a nejvyšší frekvenci, nejvyšší úroveň,... Probíhající události jsou pak takové události, jejichž aktivita byla zaznamenána při předchozím měření. Pokud nějaká událost není aktualizována, je přesunuta do instance agregátoru, kde se tyto ukončené události spojují podobně jako při detekci ve velké balíky, které jsou analyzovány.

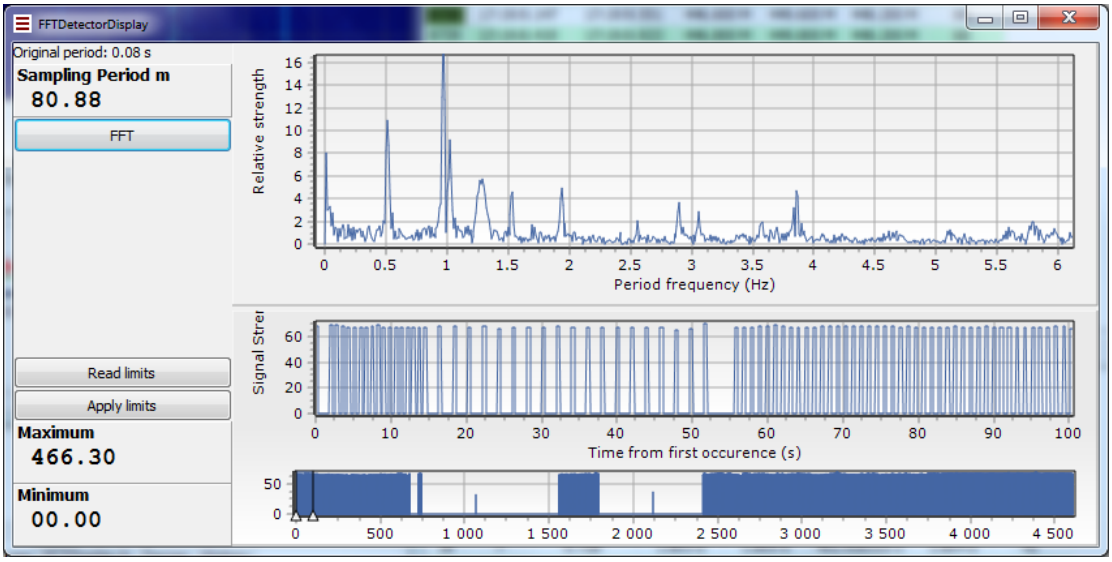
Zaznamenáváné hodnoty jsou počet událostí, ze kterých je agregovaná událost tvořena, jak dlouho trvala a kolik času byl nějaký signál zaznamenáván v poměru času mezi prvním a posledním objevením. Díky tomu je možné zjistit povahu signálu. Zda je kontinuální, nebo jde o krátké pulzy. Frekvence a úroveň jsou základní parametry a tak ani ty zde nesmějí chybět.

4.1.1 Analýza periody

Pro zjištění toho, jak signál vypadá je vhodné vizualizovat i jeho konrkétní průběh podoběn jako v Time View. A když už se má vizualizovat, je vhodné to spojit s DFFT. Na obrázku 4.3 je zachycen výsledek.

Vlevo nahoře u je `Sampling Period`. Ta se počítá jako průměrná vzorkovací frekvence. Na základě této periody je následně počítána frekvence pro výsledek DFFT. Periodu je možné změnit pro případnou kalibraci. Původní hodnota je zobrazena nad ovládacím prvkem.

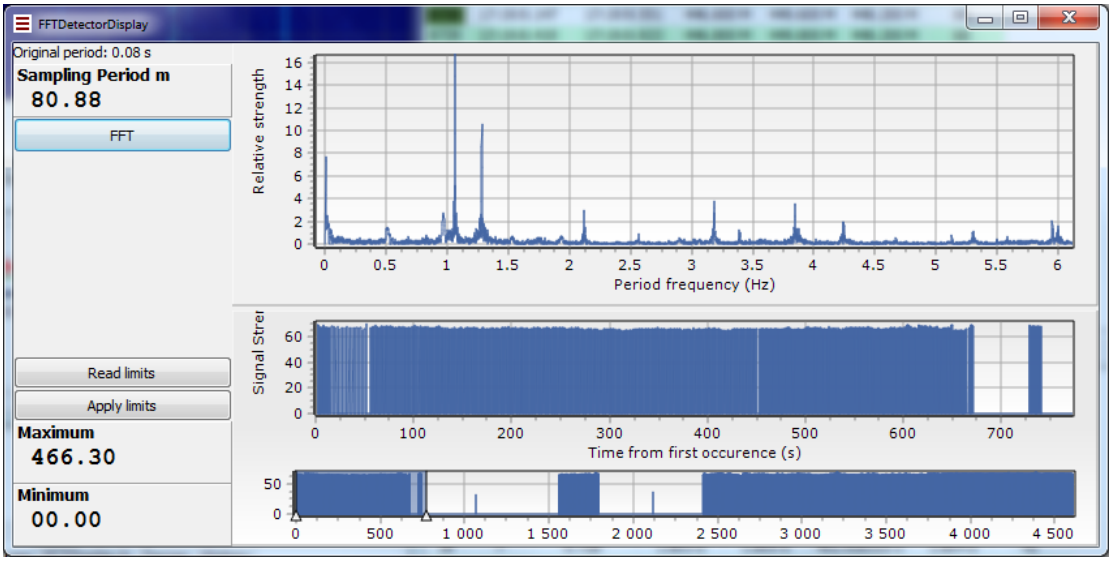
Horní graf ukazuje výsledek DFFT. Převládá signál s frekvencí 1 Hz a hned po něm je 0.5 Hz. DFFT je počítána z dat zobrazených na prostředním grafu. Spodní



Obrázek 4.3. Fourierova transformace na části událostí.

graf slouží k ovládání toho prostředního. Jsou zde dva kurzory, které slouží pro vymezení oblasti na které se analýza provádí. To může pomoci pro zpřesnění výsledků, pokud je například dlouhý výpadek, nevychází transformace tak pěkně. Případně to může být použito k omezení toho, na jaké části se analýza provádí.

V případě, že je vzorek delší, je i přesnější určení frekvencí. Na obrázku 4.4 je analyzován vzorek který obsahuje i data zobrazena na obrázku 4.3. Je vidět, že perioda se měnila a tak jsou některé frekvence velmi potlačeny na úkor jiných.



Obrázek 4.4. Fourierova transformace na jiné části vzorku.

Ovládací prvky vlevo slouží k nastavování rozsahu analýzy, protože šoupání kurzorů myší nedosahuje dostatečné přesnosti při velkém počtu vzorků.

4.2 Knihovna FFTW

Pro výpočet jsem použil knihovnu FFTW[10] se kterou mám zkušenosti z dřívějšího projektu. Do projektu se připojuje pomocí DLL. Pro snadný vývoj je pak přiložen soubor *.lib a *.h.

Vstupy a výstupy DFFT jsou struktury dostupné z hlavičkového souboru `fftwf_complex`, které musejí být alokovány pomocí `fftwf_malloc` a uvolňovány `fftwf_free`.

Samotná analýza probíhá na základě takzvaného plánu. Ten je tvořen na základě vstupních a výstupních dat, jejich délky, směru DFFT a řídících značek (flagů). První tři parametry jsou všeříkající, jediné, co je potřeba dodržet je uvedení správné velikost a použití datového typu `fftwf_complex`, který je schopen pojmout i I/Q data - komplexní data signálu, který nepoužívám. Směr udává, zda se má počítat frekvenční spektrum (`FFTW_FORWARD`), nebo rekonstruovat signál ze spektra (`FFTW_BACKWARD`). Poslední jsou značky (flagy). Ty slouží pro řízení výběru algoritmu. Mnou používaná značka `FFTW_ESTIMATE` říká plánovači, aby použil pro výběr správného algoritmu heuristiku. Jinou možností je pak například spočítání DFFT pomocí několika algoritmů a výběr toho nejlepšího. Výběr optimálního plánu tak může trvat velmi dlouho. `FFTW_ESTIMATE` je rychlý a dává rozumné výsledky - výpočet DFFT při testování nikdy netrval příliš dlouho.

Celý průběh zpracování signálu tak může vypadat takto:

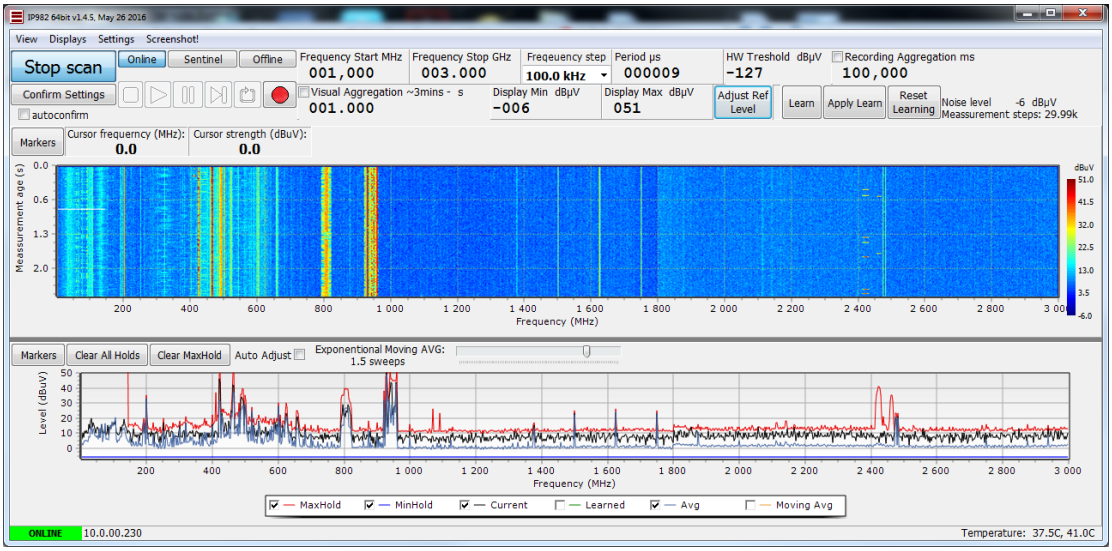
```
//příprava proměnných
int samples = 10000;
fftwf_complex *in, *out;
in = (fftwf_complex*) fftwf_malloc(sizeof(fftwf_complex) * samples);
out = (fftwf_complex*) fftwf_malloc(sizeof(fftwf_complex) * samples);

for(int i = 0; i < samples; i++){
    //naplnění vstupní proměnné vzorovým signálem
    in[i][0] = sin(i/314.15926535989)+sin(i/100)*0.7;
    in[i][1] = 0;
}

fftwf_plan p = fftwf_plan_dft_1d(samples, in, out, FFTW_FORWARD,
                          FFTW_ESTIMATE);
fftwf_execute(p);

for(int i = 0; i < samples; i++){
    //zpracování výstupu DFFT
    ... = sqrt(out[i][0]*out[i][0] + out[i][1]*out[i][1])/(samples/2);
}

fftw_free(in);
fftw_free(out);
```

Kapitola 5
Závěr

Při předělávání mě potěšilo, že ke zprovoznění na 64 bitech stačilo opravení nedostatků, jenž jsem provedl při optimalizaci návrhu.

Změna architektury a toku dat významně přispěla k rozšiřitelnosti o další způsoby zobrazení dat. Kromě toho je mnohem snazší úprava těch stávajících, díky využití integrovaného WYSIWYG a většího oddělení části pro vykreslování a zpracování dat.

Díky tomu se mi podařilo naplnit zadání. Implementoval jsem nahrávání a přehrávání. Je tak možné si pořídit záznam a kdykoliv ho analyzovat. Kombinací těchto dvou funkcionalit jsem dosáhl funkce timeshift, díky kterému je možné prohlížet si záznam, zatímco je nahráván.



Obrázek 5.1. Výsledná podoba programu.

Detekce událostí založená na překračování hodnot se ukázala jako funkční. Je možné zjistit zda se něco děje. Následnou analýzou událostí je pak možné zjistit povahu těchto událostí. Jedním z nástrojů je analýza diskrétní fourierovou transformací, kterou je možné zjistit zda je výskyt náhodný, nebo se s opakuje s nějakou periodou.

Při práci na rozšiřování programu jsem získal mnoho cenných zkušeností a ani to není to nejdůležitější - poučil jsem se z vlastních chyb, které jsem v bakalářské práci udělal. I tak ale věřím, že až se znovu pustím do rozšiřování tohoto programu, objevím nedokonalosti a chyby, které nyní nevidím. Přiznejme si ale, že pokud by se tak nestalo, musel bych být buď génius, nebo hlupák.

Literatura

[1] Embarcadero Wiki TThreadPool.
http://docwiki.embarcadero.com/Libraries/XE8/en/System.Threading.TParallel.For. Citováno dne 17.4. 2016

[2] Alexander, C.: Timing Techniques for Commodity Future Markets. McGraw-Hill, 2003.

[3] Embarcadero Wiki TThreadPool.
http://cpp-tip-of-the-day.blogspot.cz/2013/11/how-is-stddeque-implemented.html. Citováno dne 3.5. 2016

[4] Tabulka porovnání souborových systémů.
http://www.ntfs.com/ntfs_vs_fat.htm. Citováno dne 18.3. 2016

[5] Comparing Common File I/O and Data Storage Approaches.
http://www.ni.com/white-paper/9630/en/. Citováno dne 18.3. 2016

[6] Magic number (programming).
https://en.wikipedia.org/wiki/Magic_number_(programming). Citováno dne 25.3. 2016

[7] Maximum otevřených souborů.
https://msdn.microsoft.com/en-us/library/6e3b887c.aspx. Citováno dne 3.4. 2016

[8] Open and Save As Dialog Boxes.
https://msdn.microsoft.com/en-us/library/windows/desktop/ms646960(v=vs.85).aspx. Citováno dne 29.3. 2016

[9] Struktura OPENFILENAME.
https://msdn.microsoft.com/en-us/library/windows/desktop/ms646960(v=vs.85).aspx. Citováno dne 29.3. 2016

[10] Knihovna FFTW pro výpočet DFT.
http://www.fftw.org/. Citováno dne 18.6. 2016

Příloha A
Zkratky a symboly

WYSIWYG	What you see is what you get
API	Application Programming Interface
DFFT	(Discrete Fast Fourier Transform) Diskrétní Rychlá Fourierova Transformace
DLL	(Dynamic Linked Library) Dynamicky Linkovaná Knihovna
CSV	(Comma-separated values) Hodnoty oddělené čárkami
ASCII	American Standard Code for Information Interchange
SCPI	Standard Commands for Programmable Instruments

Příloha B
Obsah CD

Na CD přiložené k této práci se nalézají tyto soubory:

tex/	zdrojové soubory pro kompilaci PDF
project/	zdrojové kódy a soubory projektu
program/	zkompilovaný program spolu s konfiguračním souborem
vypracovani.pdf	elektronická verze textu této práce

Tabulka B.1. Obsah CD